TECHNISCHE UNIVERSITÄT
CHEMNITZ

Determination of Real-Time Network Configuration for Self-Adaptive Automotive Systems

Master Thesis

for
the fulfillment of the academic degree
M.Sc. in Automotive Software Engineering

Faculty of Computer Science
Department of Computer Engineering

Submitted by: Ziming Zhang.

Supervisor: Prof.Dr. W. Hardt, Dipl.-Inf.Univ. A. Stante (Fraunhofer ESK)

Abstract

The Electric/Electronic architecture of vehicle becomes more complex and costly, self-adaption can reduce the system, enhance the adaptive meanwhile reduce energy consumption and costs. The self-adaption needs the cooperation of both hardware and software reconfigurations, such that after the software is reconfigured the automotive network continues to fulfill the time constraints for time-critical applications. The thesis focuses on the real-time network reconfiguration. It uses EAST-ADL to model a real-time automotive system with timing events and constraints, which conforms to AUTOSAR timing extensions. The network media access is analyzed based on the model and a scheduling algorithm is developed. Then the concept is implemented by a use case, which is transformed from an EAST-ADL model to an executable simulation.

Keywords — Self-adapt, Dynamic network configuration, Real-time, Time-triggered, AUTOSAR timing extension, Media access control, Scheduling algorithm, EAST-ADL, OMNeT++

Contents

List of Figures

List of Tables

1. Introduction

In order to provide a better driving experience and become more environment-friendly, more and more new functions are integrated into a vehicle, which causes requirement for a more powerful automotive system. Hence the number of ECUs and the amount of software code in the system increase rapidly, which makes the system very complex and become more and more difficult to maintain. Such a situation brings great challenges to safety, reliability, energy-efficiency and cost-efficiency of the system. AUTOSAR [9] is been proved as an effective solution for such problems in the automotive industry field. Meanwhile, new ideas keep coming forth to exploit the way further, such as automotive system with self-adaptivity.
Self-adaptive functionality is been shown by many researchers as a possible solution to reduce the complexity of the system and the interactions caused by system fault and adaption handling. Self-adaptive does not mean the adaptivity just on the software level or the functional level. It is a dynamic reconfiguration of the whole system, including the hardware and software system, which are reconfigured in a synchronized way in order to keep the system running smoothly. The system should alter from one configuration to another seamlessly. The core of self-adaptive is the change of the software component, which could be changing the software allocation, shutting down certain software component or even changing part of the system functionality. The reconfiguration of the software causes new requirement to the system resource, such as hardware memory for new software, operating system resource of ECU for new tasks, time slots of the network scheduler for new data frames.
In [10, 11] typical self-adaptive scenarios are described. Dynamic software reconfiguration is a branch of self-adaptive functionality, which focuses on software maintenance on the run-time. One typical scenario is moving a software component from one ECU to another while the system is running, as shown in figure 1.1.
In the figure all the ECUs are connected to a real-time network. On the run-time ECU_2 is for certain reason down, the software components swc_2 and swc_3 are moved to ECU_1 and ECU_3 respectively, the automotive system keeps running and the communication between the moved software component not interrupted. Even the software-to-ECU allocation is changed, the system functionality keeps the same. One thing to notice is that before moving the software components, swc_2 and swc_3 are running in the same ECU, the communication between swc_2 and swc_3 is via the RTE layer in the ECU (The concept of RTE will be introduced in Chapter 2.), which won't go through the network. After swc_2 and swc_3 are moved to different ECUs, they have to communicate through the network. Such software component moving causes a problem, as shown in figure 1.2.

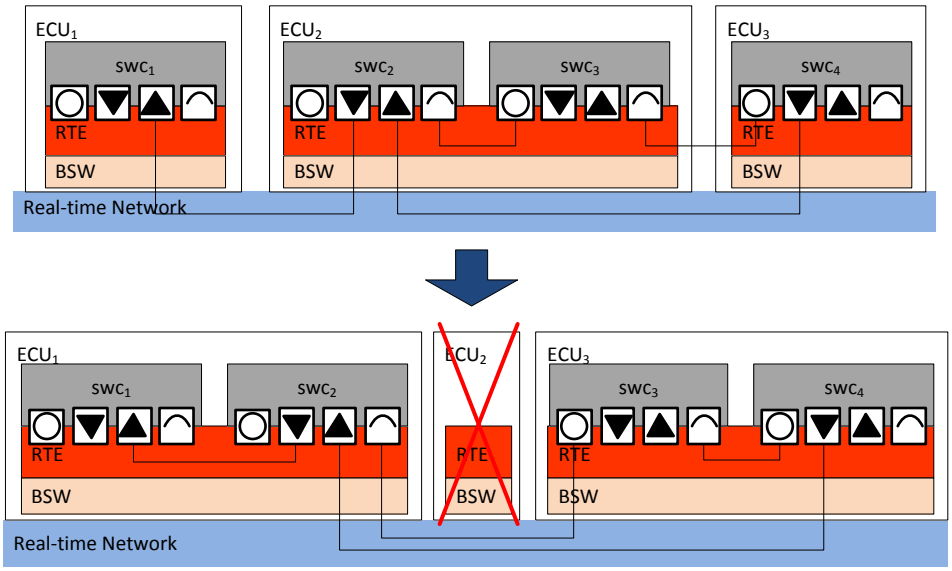

Figure 1.1.: A Typical Scenario of Dynamic Software Reconfiguration

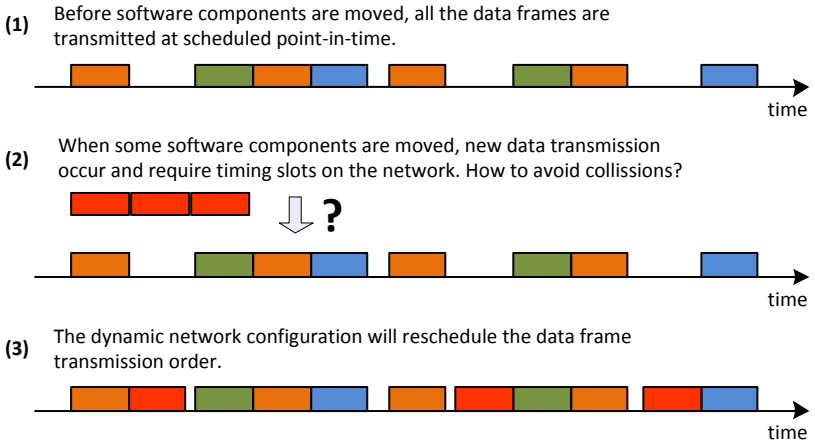

Figure 1.2.: Network Access Order Disturbed by Dynamic Software Reconfiguration

In a real-time automotive network, time-critical data is transmitted on the scheduled point-in-time, such that it won't miss the deadline. The network media access order of all the network nodes is configured previously offline, when the system is configured. After the system is initialized, all the network nodes transmit data on the network according to this fixed order. Dynamic software reconfiguration moves software components on the run-time, which may introduce new communications on the network, such as the communication between swc_2 and swc_3, which disturbs the original data transmitting order and may cause collision with other data transmissions. So there must be a mechanism which reschedules the transmitting order and avoids the collisions on the network.
To develop such a mechanism is the theme of the thesis, which is called the determi-

nation of real-time network configuration for self-adaptive automotive systems. It supports the dynamic software reconfiguration on the media access control (MAC) level. Its basic functionality is reconfiguring the network on the run-time. Reconfiguration means 'the action of changing the resources distribution including applications, partitions or communication links [10]'. On the one hand dynamic software reconfiguration requires redundant hardware resource which can be reconfigured on the run time, on the other hand the network must be configured in order to support the change in software configuration.
Dynamic hardware reconfiguration on FPGA is shown in [12, 13, 14, 15], which provides methods and strategies for dynamic reconfiguration. The following documents also provide theories for dynamic software reconfiguration strategies. In [16] a concept of reallocation of software components among ECUs is illustrated, but it doesn't support dynamic reallocation. In [11] a middle ware architecture is created to provide self-management, but it does not conform to AUTOSAR. In [17] extra services in the basic software layer in AUTOSAR architecture are created to extend dynamic reconfiguration functionality.
Meanwhile the research on real-time network reconfiguration provides theory support for dynamic reconfiguration of automotive system [18, 19, 20]. In [21] the focus is on TDMA networks, which is the most seen real-time network in automotive systems. In [22] the focus is on mapping messages into time slot in FlexRay network. In [23] the FlexRay in automotive system is implemented. Also AUTOSAR provides a methodology which is prevailing in the industry. This methodology can be extended to support dynamic reconfiguration [24, 17, 25]. In [26] a solution for real-time decision making for unmanned system is provided, which also conforms to AUTOSAR.
This thesis will provide a concept of how to determine the real-time network reconfiguration on the MAC level, which will be used for self-adaptive automotive systems. Hence it is assumed the functionality of dynamic software configuration is already realized, so the focus will be on the dynamic network configuration.
In order to configure the network access of an automotive system, first the real-time network communication is modeled and analyzed. In the model, essential timing events such as data sending and receiving should be defined. Each event has a timing constraint to control its behavior. Then based on timing events and constrains of the communication model, a scheduling algorithm for the network media access can be developed. There are mainly two kinds of dynamic reconfiguration algorithm. One is to predefine a bunch of static configurations, and dynamically select the best-fit one in the run-time, here it is called the *static configuration switch*. The other is to calculate the media access order during the run-time, find a best solution and implemented on the network, here it is called the *dynamic reconfiguration*. The two algorithms are compared in the following:

- *Static configuration switch*: Every configuration is calculated offline, so timing constraints are already validated and the software allocation and system resource usage are optimized. On the run-time the time needed for selecting

a configuration is much smaller than calculate a configuration. However, all the configurations must be defined beforehand, it becomes complicated and error-prone to cover all the necessary configurations when the system is big and complex. The flexibility is limited.

- *Dynamic reconfiguration*: The configuration is calculated according to current situation, with high flexibility. But it is time-costing to calculate and validate a network configuration, the deadline can be violated.

In this thesis the scheduling algorithm of network media access is of type *dynamic reconfiguration*, because it provides more flexibility, and as the performance of hardware becomes more powerful, the time of calculating the configuration can be managed within the deadline.
This chapter gives a brief overview of the problem and the motivation. In Chapter 2 the research fundamentals are introduced, AUTOSAR timing extensions and media access scheduling of the real-time network. Chapter 3 describes how to model the communication model and a scheduling algorithm based on the model is developed. In Chapter 4 the communication modeling concept is implemented in the EAST-ADL modeling. And a model-to-text transformation tool is developed for early phase validation of the network. In Chapter 5 a use-case is created to evaluate the implementation and the transformation tool. In Chapter 6 the the work is concluded and the outlook for future work is introduced.

2. Research Fundamentals

In this chapter, the research fundamentals are introduced. The target of the thesis is to find an AUTOSAR-conformed solution for dynamic real-time network configuration. AUTOSAR specifications which are related to function communication and the timing of methodology are introduced. And the basic scheduling algorithms and network media access control are introduced.

2.1. AUTOSAR Specifications for Modeling Function Communication

AUTOSAR (AUTomotive Open System ARchitecture) is an open industry standard for automotive E/E architectures, which aims to manage increasing E/E complexity, improve flexibility of for product modification, improve scalability of solutions, improve quality and reliability of E/E systems and enable detection of errors in early design phases [1]. The basic AUTOSAR approach is shown in figure 2.1.

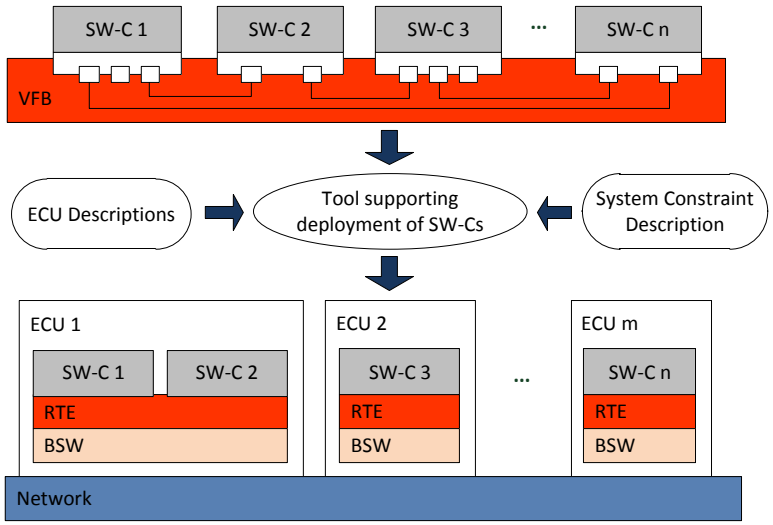

Figure 2.1.: Basic Approach of AUTOSAR [1]

The functionality of an automotive system is realized by multiple software components. A software component (SW-C) is a piece of software that implements part of an application. It is independent from the system infrastructure, and can be mapped on an ECU. If a SW-C can not be distributed over several ECUs, it is

called atomic software component, otherwise it is a composition. The concept of software component is introduced in detail in Subsection 2.1.2. Sensor/actuator software components are special software components, which are mapped to ECUs directly connecting its corresponding sensor/actuator.
The communication relations of all the software components are defined in the VFB view. VFB is short for virtual functional bus, which is the sum of all communication mechanisms of the system. It allows a virtual integration of software components on a technology independent level. The concept of VFB is introduced in detail in Subsection 2.1.2.
AUTOSAR defines a methodology to develop an automotive system, the integrated software components are mapped to ECUs and the automotive network is configured. The software architecture of an ECU is divided into three layers, the software components are on the top layer, the run-time environment (RTE) layer implements the VFB functionality on the ECU, the basic software (BSW) layer provides the infrastructural functionality on an ECU. The AUTOSAR architecture will be introduced in detail in Subsection 2.1.1.

2.1.1. Software Architecture of ECU

AUTOSAR defines a three-layer software architecture of an ECU. The first layer is the Application Layer, which is composed of software components. These components are mapped to ECU by the AUTOSAR Methodology. Application is a specific functionality of the system, such as break-by-wire (BBW) or automatic cruise control (ACC), which can be further divided into software components.
The second layer is the RTE, which exchanges information between ECUs. As the communication requirements of the software components are application dependent, the RTE needs to be tailored partly by ECU and partly by software components running on it.
The third layer is the BSW Layer. BSW is the standardized software layer, which provides services to the software components. The BSW layer can be further divided into three sub-layers. Services layer offers operating system functionality, communication services, memory services, diagnostic services and so on. ECU abstraction layer offers an API to access peripherals. And microcontroller abstraction layer contains the drivers of microcontroller and internal peripherals. It makes higher software layers independent from the microcontroller.
Each sublayer can be further divided into module groups. Fox example, the service layer contains system services, memory services and communication services. The module groups in different sub-layers form the so-called ‘Stack’. For example, the Communication Services, the Communication Hardware Abstraction, the Communication Drivers and I/O Drivers are called the Communication Stack, which supports vehicle network communication such as CAN, FlexRay, Ethernet and so on.

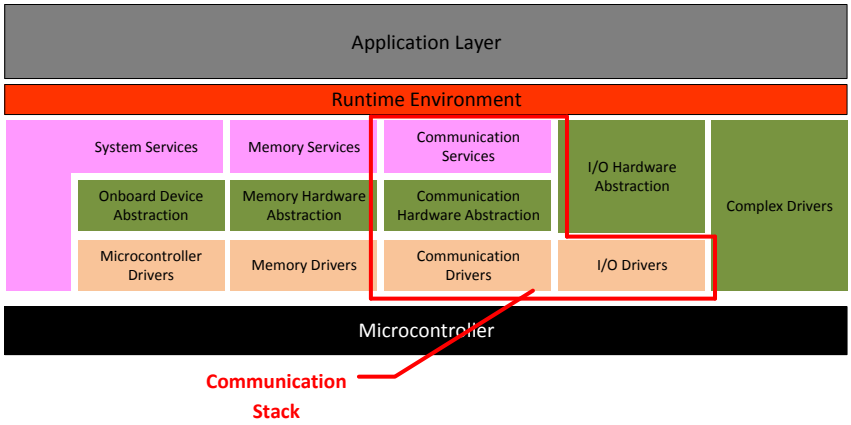

Figure 2.2.: Three-layer Software Architecture [2]

2.1.2. Virtual Functional Bus

A vehicle system is composed of various functions, such as anti-lock braking system (ABS), steering-by-wire, seat heating etc. In AUTOSAR, such a function is defined as an application, which consists of multiple interconnected software components. On the VFB level, the structural element to build a system is software component. VFB is the communication mechanism that allows these components to interact. During the system configuration, the components are mapped on ECUs and the component connections defined in VFB are mapped either within a single ECU or on the communication network such as CAN, Ethernet or FlexRay. A software component has ports, through which a software component communicate with other software components. A software component can have multiple instances in an application. The instance is called prototype. An example of application *Seat Heating* [3] is illustrated by figure 2.3. Ports which do not concern communication, such as *EcuMode* or *Calibration* are not shown in the figure.
The functionality of the application is fulfilled by a group of software components. *SeatSwitch* detects if a passenger is sitting on the seat and informs *SeatHeatingControl*. *HeatingDial* sends desired seat temperature to *SeatHeatingControl*. *SeatHeatingControl* reads the temperature value and sends it to *SeatHeating*. *SeatHeating* receives desired heating setting and controls the seat heating hardware. In certain conditions *PowerMangement* could decide to disable seat heating to avoid accident.
A software component is either an atomic software component, which cannot be divided further into smaller components, or a composition. Multiple connected software components form a sub-system for a specific usage. The sub-system can be packaged as a component, called software composition. An atomic software component consists of one or more runnable runnables. A runnable is a sequence of instructions which can be executed at run time, it can be either a very small code piece of some simple algorithm or a complex program. A composition has its own ports. The software components *HeatingDial*, *SeatHeating* and *SeatHeatingControl*

can compose a composition *SeatHeatingControlAndDrivers*.

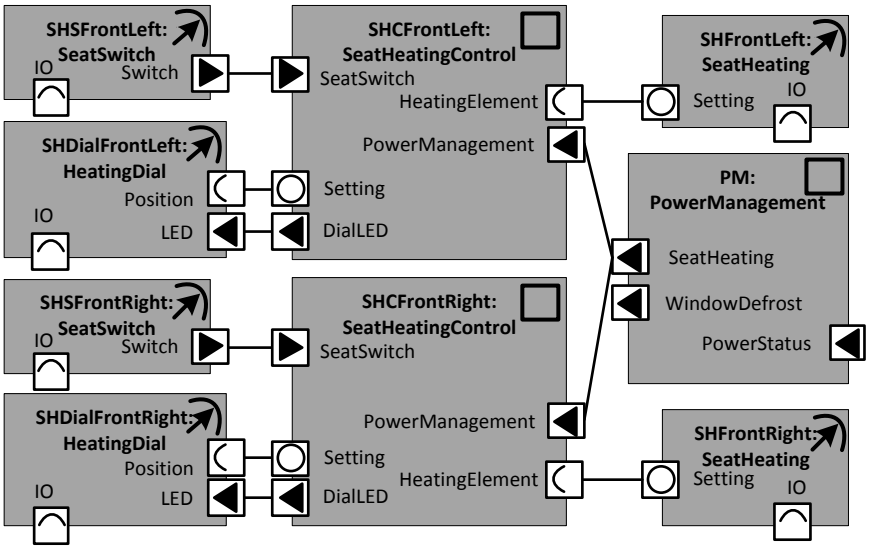

Figure 2.3.: Seat Heating Application [3]

The software components are connected by ports. A port of is either a providing port (pport) or a receiving port (rport). A pport provides the elements defined in a port-interface. An rport requires the element defined in a port-interface. A port is associated with a port-interface. Interface defines the communication contract of the port. There are sender-receiver interface and client-server interface.
A sender-receiver interface defines data-elements that are sent and received over the VFB. Senders can send out data with or without a receiver. The interface between port *Switch* and *SeatSwitch* is such an example. For sender-receiver communication one pport can be connected to one or more rports, which is multicast. And one or more pports can be connected to one rport, collecting information from different senders in a single receiver.
A client-server interface defines operations that can be invoked by a client and implemented by a server. The server only provides the service if there is the request from a client. The interface between port *HeatingElment* and *Setting* is such an example. For client-server communication one pport can be connected to one or more rports, i.e. multiple clients can ask for service from a single server. But one rport connects to more pports is not allowed, i.e. a client cannot receive service from multiple servers.
When an application is deployed on a network of ECUs, the atomic software components are mapped to ECUs and assembly-connectors are implemented as communications which are either inside an ECU or between ECUs on the network. The communication paradigms are provided by RTE.

2.1.3. Timing Extensions of Methodology

In AUTOSAR timing extensions the basic entity is event, which is used to refer to an observable behavior within a system at a certain point-in-time. The occurrence of events and their exact timing and the concurrency of various events are important to specify the system behavior. The purpose is to provide timing requirements that guide the construction of systems and to provide sufficient timing information to analyze and validate the temporal behavior of a system [4].
The timing extensions serve as timing specification in the AUTOSAR methodology. In AUTOSAR, the concept of methodology provides an outline of design steps to build an automotive system. There are five different timing extension views in the design phases of methodology, as shown in figure 2.4.

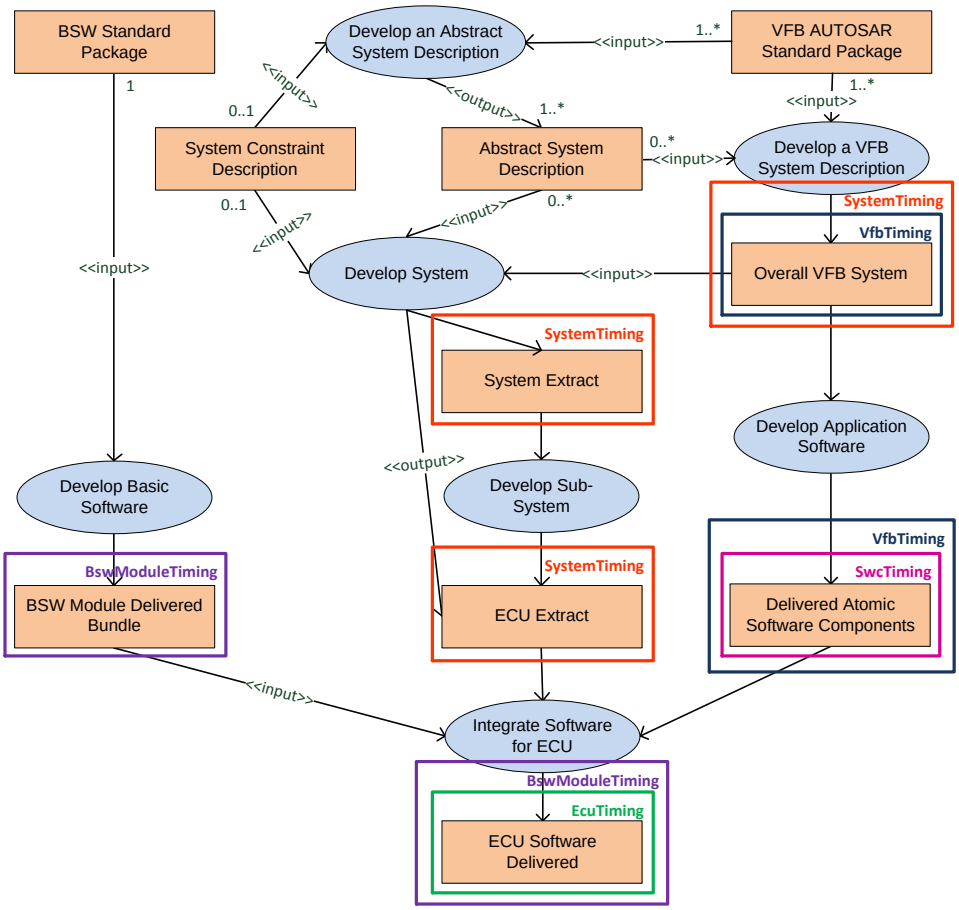

Figure 2.4.: Five Timing Views in AUTOSAR Methodology [4]

The *Abstract System Description* represents the overall system from a functional architecture view. This is the basic of development of concrete system description. The *Overall VFB System* provides a software architecture of all the functions. The *VFBTiming* and *SystemTiming* extensions are in this phase. which is then refined to the Overall System. *VfbTiming* deals with the interaction of software components

regardless the underlying physical system. Software components are treated as black boxes, their internal behaviors are not considered. The events at this level only refers to software components, ports and their connections. The events can be used in other timing views, including *SwcTiming*, *SystemTiming* and *EcuTiming*. *SystemTiming* deals with the hardware topology, software deployment and communication matrix. The abstract communication between software components in *VfbTiming* becomes concrete here. The events related to *SystemTiming* can be used in *EcuTiming*.
The *System Extract* represent sub-systems of the whole system. The sub-systems contains VFB reduced from the system VFB. The *ECU Extract* is extracted from the system description. It is related to a specific ECU. The deliverable of each ECU is based on the ECU Extract. The *SystemTiming* is also in this two phases.
While the system is designed, the *Delivered Atomic Software Components* are developed based on VFB description. They are then integrated with *ECU Extract*s into the ECU on which they are deployed. The *VFBTiming* and *SwcTiming* extensions are in this phase. *SwcTiming* deals with the internal behavior of atomic software components. The events related to *SwcTiming* can be used in *SystemTiming* and *EcuTiming*.
The basic software is independent from VFB, hence the *BSW Module Delivered Bundle* is developed at any time before the ECU software integration. The *BswModuleTiming* is in this phase. It deals with the internal behavior of a single BSW module, such as the activation, start and end of a BSW module entity. The events related to *BswModuleTiming* can be used in *EcuTiming*.
After the software has been integrated into ECU, the methodology reaches the last phase, *ECU Software Delivered*. The ECU is configured by creating tasks, scheduling runnables, configuring basic software. The complete code is compiled and linked into an executable. The *BswModuleTiming* and *EcuTiming* are in this phase. *EcuTiming* can be considered as a *SystemTiming* focusing on one specific ECU. It deals with the deployed software components, the ECU interactions and the basic software.
There are three abstract types. The formal basis is *TimingDescriptionEvent*, as event is the basic entity. A system's timing behavior usually does not only include single events, but also the correlation of different events, which is called *TimingDescriptionEventChain*. The event chain is used to describe the order of dependent events. The start of the chain is a stimulating event named *sitmulus*, the end of the chain is a responding event named *response*. An event chain can be further divided into a set of sub-chains, called *segments*. Based on events and event chains, their timing behaviors are expressed by *TimingConstraint*. The timing constraints are independent from implementation details. There are eight types of timing constraints, including *Event Triggering*, *Latency Timing*, *Age*, *Synchronization Timing for event chains*, *Synchronization Timing for events*, *Offset Timing*, *Execution Order* and *Execution Time*. The event triggering constraint describes the occurrence of an event. The occurrence pattern of an event could be periodic, sporadic, burst, concrete pattern or arbitrary. The events which are related to software component communication are shown in figure 2.5.

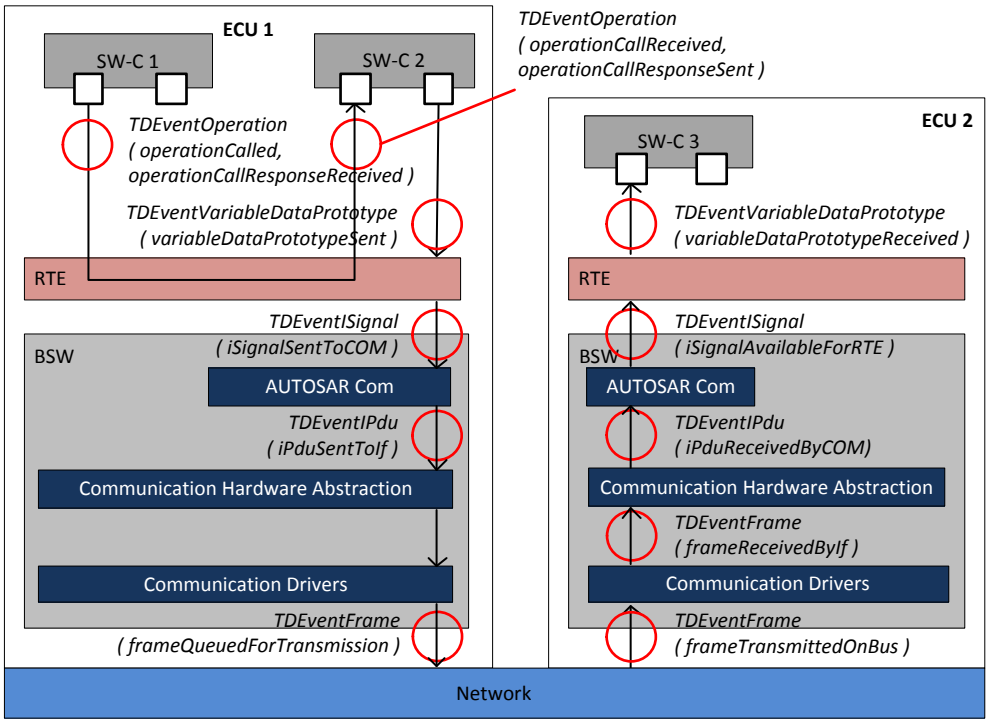

Figure 2.5.: TimingDescriptionEvents of Software Component Communication

TDEventVariableDataPrototype describes events related to sender-receiver communication on the function port. There are two types of events. *variableDataPrototypeReceived* is the event when the data is available in the RTE for the software component. *variableDataPrototypeSent* is the event when the data is available in the RTE for transmission.
The *TDEventOperation* describes events related to client-server communication on the function port. There are four types of events. *operationCalled* is the event when the operation is called by the client software component. The call will be received by the server software component, the event is named *operationCallReceived*. The server software component then finishes the operation and sends out a response, the event when the response is sent is named *operationCallResponseSent*. *operationCallResponseReceived* is the event when the client software component has received the response of the operation call.
The data from the function ports will be packed into I-Signal in RTE and sent to the BSW layer. *TDEventISignal* describes events related to I-Signal exchange between the RTE and the *AUTOSAR Com* sublayer in the BSW. There two types of events. *iSignalSentToCOM* is the event when the RTE has sent out a communication signal to the I-PDU buffer in the *AUTOSAR Com*. *iSignalAvailableForRTE* is the event when the *AUTOSAR Com* is ready with a communication signal for the RTE.
The I-Signals will be packed into I-PDUs in *AUTOSAR Com* and sent to *Communication Hardware Abstraction* sublayer in the BSW. PDU is short for protocol data unit.

TDEventIPdu describes events related to I-PDU exchange between these two sublayers. There are two types of events. *iPduSentToIf* is the event when the PDU is pushed to the bus interface. *iPduReceivedByCOM* is the event when the PDU is pushed from the bus interface to the AUTOSAR Com.
The I-PDUs will be packed into frames in *Communication Hardware Abstraction* and sent to *Communication Driver* sublayer in the BSW. *TDEventFrame* describes events related to Frames exchange between the specific bus interface module and the communication controller of the *Communication Drivers* sublayer in the BSW. There are three types of events. *frameQueuedForTransmission* is the event when the PDUs are queued in the frame in the communication driver. *frameTransmittedOnBus* is the event when the frame is transmitted to the communication controller of the subscriber. *frameReceivedByIf* is the event when the frame is pushed from the communication controller to the specific bus interface module.

2.2. Media Access Control in Real-time Network

A real-time network transfers messages within a constant transmission delay and bounded jitter. There are several ways to access the network media for real-time networks, such as token-based, time/frequency/code- division multiple access (TDMA/FDMA/CDMA) or bandwidth limiting. In automotive systems, real-time network is based on TDMA mechanism. Currently the widely implemented is FlexRay [27], which is a high-bandwidth TDMA bus system. TTCAN [28] (ISO 11898-4) is an extension of CAN bus, which transfers CAN messages in a time-triggered way. Ethernet is one of the most widely implemented network standards, which also plays a more and more important role in the automotive industry now. Meanwhile real-time Ethernet for time critical applications has been researched deeply. [29, 30] give a list and introduction of different industrial real-time Ethernets. Among them, 802.1 AVB [31], 802.1 TSN [32] and TTEthernet [33] are implemented in automotive systems. Now a concept of 'Global Time Synchronization [34]' is proposed by AUTOSAR, which tries to synchronize different type of networks in the automotive system 'in order to precisely recognize the actual surroundings [34]'.
By the principle of message transfer, network communications can be divided into time-triggered and event-triggered. In an event-triggered system, communication occurs when some events happens, it is sporadic. In a time-triggered system, messages are initiated based on the progression of time. TDMA is the most implemented medium access mechanism in the automotive real-time network. Each node is assigned a time slot in a static time segment and transfers messages. Hence there is no extra time for arbitration or avoid-collision retreat in the network. But there is no explicit definition of network nodes scheduling in those network specifications, which is solved in the design. All the network nodes are synchronized. Without a synchronized clock, The nodes can't be scheduled to access medium exactly. For example AVB, TSN and TTE all implement IEEE 1588 [35] for synchronization. With the synchronization method fault nodes can be filtered out of the network.

2.2.1. Basic Scheduling Algorithms

Scheduling means allocating resources to concurrent users. The resources can be CPUs, memories, hard disks or buses. From the view of a scheduler, the basic unit to be planned and implemented is called a job. In a real-time system, the scheduler must make sure all the scheduled jobs can be finished within the deadline. If no deadline is missed, the schedule is considered to be implementable. The timing parameters which are used to describe the execution state of a job is shown in figure 2.6.

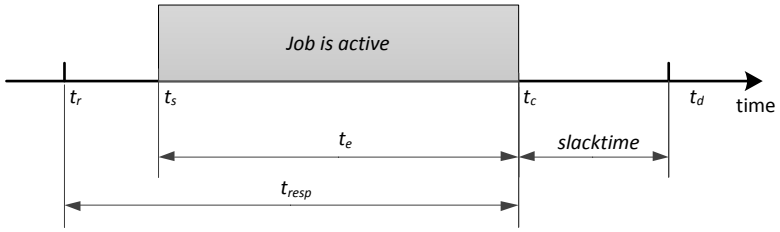

Figure 2.6.: Timing Parameters of A Job [5]

The point-in-time when a job is ready to be executed is called the release time, t_r. There is usually a waiting list in a scheduler. A job will be put in the waiting list at t_r by the first-in-first-out (FIFO) principle. After the jobs queued in front are executed, this job will be removed from the waiting list and start at a scheduled point-in-time, called the starting time, t_s. The point-in-time when the job is finished is called the completion time, t_c. There is the point-in-time, before which t_c must occur, this is the deadline D. If t_c occurs after D, the deadline is missed. From t_r to t_c, the time duration is called the response time t_{resp}. If the job is executed without interruption, the time duration from t_s to t_c is called the execution time e. Normally the response time is longer than the execution time. For a time-critical job, t_{resp} can be used as the worst-case execution time of the job, it makes more sense than e. The time duration from t_c to t_d is called the slack time *slacktime*.
The i_{th} periodical job can be defined as $J_i(P_i, e_i, D_i)$. Usually the period of the job is used as its deadline, so J_i is described as $J_i(e_i, D_i)$. When J_i is executed it occupies the resource of the system, which is called the 'load'. The load of J_i is defined as:

$$U_i = \frac{e_i}{D_i} \quad (0 \leq U_i \leq 1) \tag{2.1}$$

The load of the system is defined as:

$$U = \sum U_i = \sum \frac{e_i}{D_i} \quad (0 \leq U \leq 1) \tag{2.2}$$

The most seen online real-time scheduling methods include:

- RoundRobin: There is no priority difference between the jobs, every job is executed in a fixed order one after another.

- Earliest Deadline First (EDF): The priority of the job is variant. The job which has the shortest deadline has the highest priority thus can preempt other jobs. The condition of implementing EDF is the system load satisfies $U \leq 1$.
- Rate Monotonic Scheduling (RMS): The priority of a job is related to its period, the shorter the period is, the higher the priority is. Because period is used as the deadline, it is can be considered which has the shorter deadline has the higher priority. But the period of a job is fixed, so unlike EDF, the priority of a job is invariant. The condition of implementing RMS is the system load satisfies $U \leq n(2^{\frac{1}{n}} - 1)$ ($U_\infty \approx 0.693$).
- Minimum Laxity First (MLF): The priority of the job is variant. The job which has the shortest slack time has the highest priority thus can preempt other jobs.

2.2.2. Network Access Scheduling

The model of a real-time network is shown in figure 2.7. In the figure, network nodes have data waiting in their sending lists, the scheduler decides which one can send data on the network. In the network, the reaction begins from the income of a network node and ends at the outcome of another node, the expected result should be available at a particular point-in-time. The reaction time of the communication includes the reaction time of each node and the transmission time between the nodes.

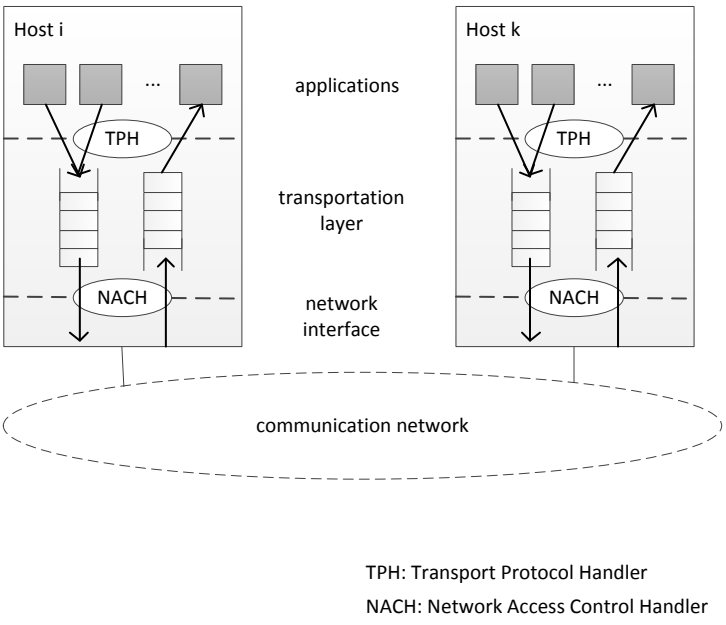

Figure 2.7.: An Example of Real-Time Network [6]

The unit of the data transmitted on the network is called packet or frame, which has a fixed maximum size. If the data from a node is too large, it will be divided into multiple packets for the transmission. The performance of a real-time network can be measured by several values. First is the transmission delay. It starts from the sending task of the node and ends at the receiving task of another node. It is an essential parameter for a real-time network. There is a expected arrival time of packets for periodical communication, but the actual arrival time is usually sooner or later, the difference is called jitter. For time-critical system, too large jitters are intolerant. One solution to decrease jitter is using lager buffer, but it will increase transmission delay.
Another important parameter is called the throughput. It is the number of packets transmitted on the network in a time unit. The lager the number is, the higher speed the network has. For media access scheduling algorithm, the minimum throughput is required. Only with throughput is can not provide enough, two more parameters are used to describe the quality of the transmission. One is the the rate of the missed deadline and damaged messages. The value of the highest tolerant rate is set as the performance requirement and the number of the missed deadline and the damaged messages are counted, the two numbers are compared. The other the duration of the interruption, which is the maximum number of continuous damaged messages.
There are two types of media access policies. One is random-access. The nodes must compete for data transmission, arbitration or collision-recognition algorithm is implemented for such access. The random-access mechanism is implemented for networks such as LAN, WLAN or CAN. The other is central-master. In the system, there is a fixed list, which saves the access order of all the nodes. TDMA is typical example of central-master. The transmission time are divided into time slots and allocated to the network nodes, each node can only send data at its own time slot. In order to keep all the nodes act according to identical clock, extra synchronization algorithm is needed.
In a real-time network, if the packet-sending action of a network node is treated as a job, the network media resource is treated as the CUP resource, the scheduling algorithm of real-time system can be implemented for network media access.

3. Function Communication Model and Determination of Network Configuration

In the automotive network, a network node is an ECU. The network media is shared by all the network nodes. When the ECUs transmit time-critical data, they must conform to certain media access rules to avoid transmission collision. The data is generated by functions which are allocated to ECUs. An ECU usually contains multiple functions. If a function sends out data to another function which belongs to another ECU, its own ECU will encapsulate the data into a data frame and transmit the frame on the network to the other ECU. Whether an ECU transmits a data frame is decided by whether its functions generate data, hence to control the network access of ECUs is actually to control the network access of functions. To analyze the network media access behavior of functions, the communication between functions should be analyzed first.
In the first section, the communication between two functions is modeled, the timing events and constraints which are essential to describe the media access behavior are defined in the model. Based on the communication model, a scheduling algorithm for the network media access is created in next section.

3.1. Function Communication Model

On the analysis level of EAST-ADL, a function is equal to an AUTOSAR-defined application, such as braking-by-wire (BBW) or adaptive cruise control (ACC), which realizes a specific functionality of the automotive system. If an application is time-critical, its reaction deadline must not be violated. For example the BBW should react within $5ms$ as soon as the drive hits the brake, otherwise it may cause serious safety accident.
A reaction can be expressed by a chain of related events, there are at least two events in this chain, a stimulus event and a response event. Event is an action which changes the state of the system. The range of an application reaction can be various by selecting different events. By considering the affection of the environment, the stimulus event of an application could be that a sensor senses the change of the environment, and the response event is that the expected action is executed. If the focus is on the application itself, regardless the environment, such as sensor/actuator which generates/consumes data, the stimulus event could be that the application is

triggered by the data from the sensor which senses the action of the vehicle driver. The response event is the application sends out data to the actuator which indicates the machine to finish the expected actions. Sensor/actuator is connected directly to the ECU, not through the network, the data transmission between them won't occupy any media resource. Hence omitting sensor/actuator in the communication model won't affect the media access control.
When and how often an event occurs is defined by its timing constraints. The event chain also has its timing constraints, which defines the time delay between two related events. The value of the reaction delay is decided by the requirement defined by the system designer.
When the application is mapped from the analysis level to the design level, it could be divided into multiple connected functions. For example on the design level the BBW is divided into functions such as brake-torque calculator, brake controller and multiple ABS controllers. The functions are connected through rports and pports, so an application can be considered as a chain of functions. Each function has its own reaction delay, the total reaction delay of the function chain is the reaction delay of the application.
In order to describe the communication behavior of between functions on the design level, a communication model is created. No matter what topology the connected function have formed, the communication behavior on every connection is the same, because all the communications conform to the same network protocol. So a model with two connected functions will provide enough information for communication analysis.
In the the model there is an application named $application_k$, it is one of the applications in an automotive system. $application_k$ contains two functions, $function_a$ and $function_b$. $function_a$ and $function_b$ are connected through sender-receiver ports. $function_a$ sends out data to $function_b$ by a pport and $function_b$ receives the data with a rport. If $function_a$ and $function_b$ are allocated to the same ECU, the communication between them is on the RTE, which won't occupy any network resource. In this thesis such a communication is called the intra-ECU communication. Otherwise if the two functions are allocated to two different ECUs, the communication between them is on the network, such a communication is called the inter-ECU communication.

3.1.1. Communication Model on VFB Level

The two connected functions which are allocated to the same ECU is illustrated in figure 3.1. The red circles in the figure illustrates that the events occur on the flow ports. The event on the rport is the data-receiving event, the event on the pport is the data-sending event.
The point-in-time when the events occur and the reaction constraints between the events are illustrated in figure 3.2, the events are listed on the time axis:

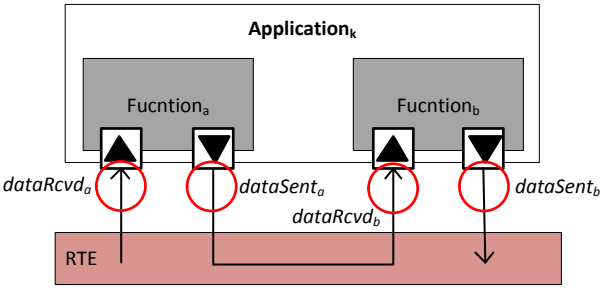

Figure 3.1.: Intra-ECU Communication between Two Functions

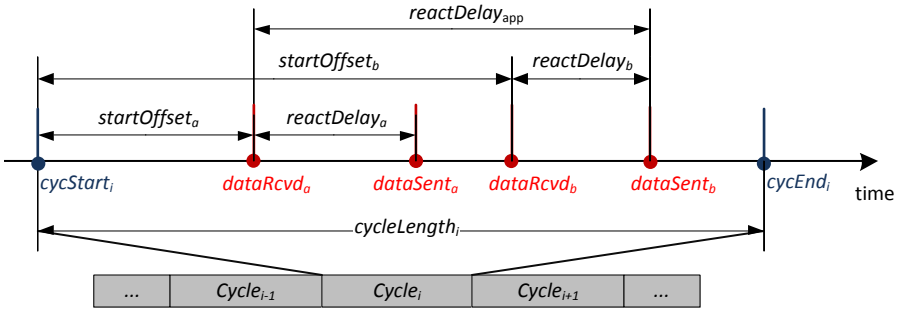

Figure 3.2.: Events and Constraints of Intra-ECU Communication

The figure illustrates the relations of the events in the i_{th} communication cycles. For real-time network, each network nodes has its own data-sending period, the communication cycle is defined as the least common multiple (LCM) of all data-sending the periods. On the time axis, $cycStart_i$ is the start of the i_{th} communication cycle, which can be seen as a original point on the time axis, the location of every event on the time axis within the cycle can be indicated by this point. $cycEnd_i$ is the end of the i_{th} communication cycle, its position is decided by the length of the communication cycle. $function_a$ and $function_b$ are time-triggered functions. Each has its own data-sending period, $sendPeriod_a$ and $sendPeriod_b$. $cycleLength_i$ is the length of the i_{th} cycle, its value is calculated by equation (3.1).

$$cycleLength_i = LCM(sendPeriod_a, sendPeriod_b) \tag{3.1}$$

$function_a$ and $function_b$ also have the data-receiving periods, $rcvPeriod_a$ and $rcvPeriod_b$. $function_b$ receives data from $function_a$, so the value of $rcvPeriod_b$ is decided by $sendPeriod_a$, as defined in equation (3.2). Both data-sending period and data-receiving period of a function are properties of the function, which are defined by the function designer according to the design requirement and the functionality of the function.

$$rcvPeriod_b = sendPeriod_a \tag{3.2}$$

The first event $dataRcvd_a$ occurring on the time axis is the data-receiving event

at the rport of $function_a$. It is a time-triggered event and occurs with the period $rcvPeriod_a$. $dataRcvd_a$ is considered as the start point of $function_a$'s reaction. The $cycStart_i$ is set as the original point on the time axis in the i_{th} communication, the point-in-time when $dataRcvd_a$ occurs can be expressed by an offset from $cycStart_i$, defined as the $startOffset_a$. The value of $startOffset_a$ indicates the expected occurring point-in-time of $dataRcvd_a$, it is a property of the function, which should be defined by the function designer according to the design requirement and the functionality of the function.
The next event $dataSent_a$ is the data-sending event which occurs at the pport of $function_a$. It is also time triggered, and occurs with the period $sendPeriod_a$. It is considered as the end of $function_a$'s reaction. The point-in-time when $dataSent_a$ occurs is decided by $dataRcvd_a$ and $reactDelay_a$, as shown in equation 3.3. $reactDelay_a$ is the reaction delay of $function_a$, which will be described later in this subsection.

$$dataSent_a = startOffset_a + reactDelay_a \tag{3.3}$$

The next two events belong to $function_b$, the data-receiving event $dataRcvd_b$ on $rport_b$ and the data-sending event $dataSent_a$ on $pport_b$. The same as $function_a$, the point-in-time when $dataRcvd_b$ occurs is $startOffset_b$, i.e. the offset from $cycStart_i$. It is a property of $function_b$ and should be provided by the function designer. The point-in-time when $dataSent_b$ occurs is decided by $dataRcvd_b$ and $reactDelay_b$, the reaction delay of $function_b$, as shown in equation 3.4.

$$dataSent_b = startOffset_b + reactDelay_b \tag{3.4}$$

$application_k$ is composed of $function_a$ and $function_b$. The application has a reaction delay $reactDelay_{app}$. The length of $reactDelay_{app}$ is decided by the design requirement of the application. $reactDelay_{app}$ can be expressed by the events on function's flow ports. The stimulus event of $reactDelay_{app}$ is $dataRcvd_a$, i.e. when $function_a$ is triggered, the application is triggered to react. The response event of $reactDelay_{app}$ is $dataSent_b$, i.e. when $function_b$ ends the reaction, the application ends the reaction. From the view of design requirement on the vehicle level of EAST-ADL, only the application reaction delay is concerned, the functions which consist the application are not considered. For example, the requirement defines the reaction delay of BBW should be no longer than $5ms$, but no consideration will be given to how much time it takes for functions such as the brake controller or torque calculator to react. On the contrast, in order to evaluate the performance of the system on the design level, the point-in-time when a function is triggered and the reaction deadline of the function should be defined. The application's reaction constraint provides a basic limit to set the timing constraints of each function. The value of $reactDelay_{app}$ should be provided by the system designer.
$function_a$ has a reaction delay $reactDelay_a$. The stimulus event is $dataRcvd_a$ and the response event is $dataSent_a$. $reactDelay_a$ is a time limit within which the execution of $function_a$ must be finished. The reaction delay of a function is different from the execution delay. The execution delay of a function only contains the

time delays when the function is executed. The execution might be interrupted by other tasks in the operating system of the ECU, the reaction delay also include the interruption delays. Hence reaction delay is normally longer than the execution delay and is difficult to estimate. A worst-case reaction delay can be estimated by the function designer, which can be used as a property of the function for an early-phase validation. This worst-case delay can be used as the reaction delay of the function here. It is a deadline that can not be violated.
$reactDelay_b$ is the same to $reactDelay_a$, which is the reaction deadline of $function_b$. Between $reactDelay_a$ and $reactDelay_b$ there is a blank on the time axis, which means the $reactDelay_b$ might not begin as soon as $reactDelay_a$ is finished. The $dataRcvd_b$ can occur at anytime as long as the $reactDelay_b$ won't violate the deadline of the application.

3.1.2. Communication Model on System Level

The two connected functions which are allocated to different ECUs is illustrated in figure 3.3. $function_a$ and $function_b$ are allocated to ECU_1 and ECU_2 respectively, the communication is then from RTE onto the network.

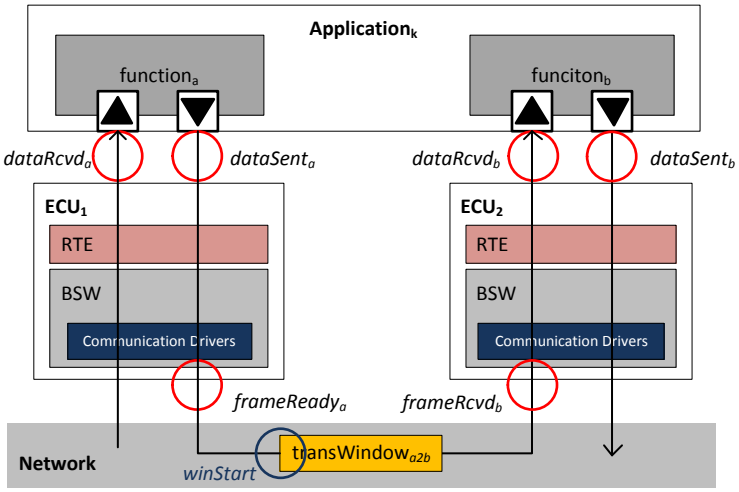

Figure 3.3.: Inter-ECU Communication between Two Functions

In figure 3.1 the timing events on the flow ports of functions are defined, which are not enough to express the data transmission from RTE to network. AUTOSAR timing extension also defines timing events on the BSW layer of an ECU, among which the data frame queuing and sent events can be used for describing the transmission on the network, as shown in figure 3.4:
The new event $frameReady_a$ is the event that the data frame is queuing in the communication driver of ECU_1 and waits for transmission. When the data is sent out from $function_a$'s pport, it goes through RTE to the BSW layer of ECU_1, the data will be packed into a data frame and finally reaches the ECU_1's communi-

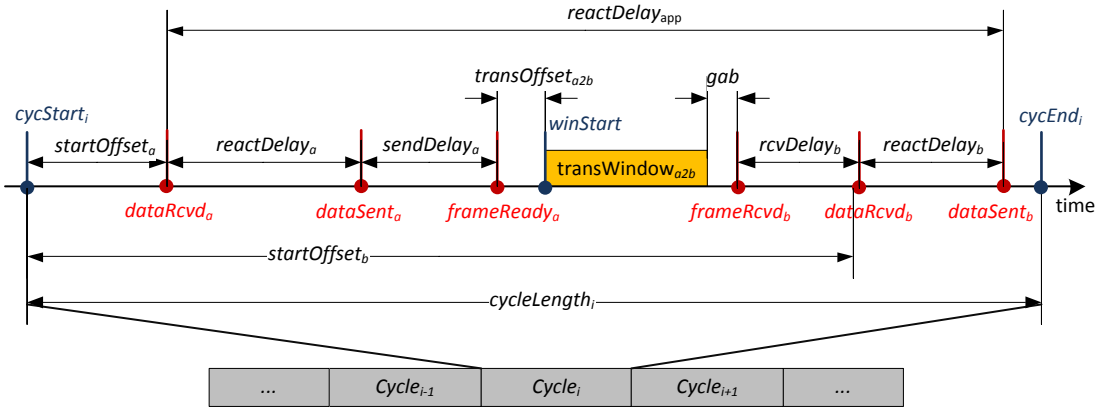

Figure 3.4.: Events and Constraints of Inter-ECU Communication

cation driver. In the driver the frame waits until the scheduled point-in-time for transmission, then it be transmitted on the network to its destination.
It costs time from $dataSent_a$ to $frameReady_a$ to transmit data and pack it into a frame, so there is a reaction delay $sendDelay_a$, which is related to the property of the ECU. The delay value is difficult to estimate, because it can be interrupted by other tasks in the ECU, and it also depends on the ECU's performance. But a worst-case delay can be estimated in order to validate the system performance in the early phase of design. With $sendDelay_a$ the point-in-time when $frameReady_a$ occurs can be calculated by equation 3.5.

$$frameReady_a = dataSent_a + sendDelay_a \tag{3.5}$$

When the data frame has reached the communication driver, it won't be transmitted on the network immediately. In a TDMA network, when a data frame is transmitted is scheduled by the media access controller, it must wait for its own time slot, which is allocated by the scheduler. So there is a time delay from $frameReady_a$ to the point-in-time when the frame is actually transmitted. This offset from $frameReady_a$ is defined as $tranOffset_{a2b}$, which depends on the connection between functions. If a function has multiple connections to different functions, the value of the offset on each connection will be different from each other. This offset value is the key to set the transmitting time of every data frame in order to avoid transmission collisions. The scheduling algorithm of network media access is in charge of calculating this value.
When the data frame is finally transmitted on the network, the transmission duration is various, depending on the size of the frame. In this thesis, this transmission duration is called a transmitting window. There is no such an event in AUTOSAR timing extension which indicates the start of the transmitting window, but the point-in-time of the start can be calculated by the transmitting offset, as shown in

equation 3.6.

$$twStart = frameReady_a + transOffset_{a2b} \tag{3.6}$$

The $transWindow$ is the length of the transmitting window, which is decided by the size of the data $dataSize_a$ and the transmitting speed of the network bus $busSpeed$. The equation 3.7 is used for calculating $transWindow$.

$$transWindow = \frac{dataSize_a}{busSpeed} \tag{3.7}$$

When the data frame has reached ECU_2, it is received by the communication driver of the ECU. This event is named $frameRcvd_b$. Then the frame goes up through the BSW layer. It is unpacked, the data which is destined to $function_b$ goes through the RTE and reaches the rport of $function_b$. There are two ways to calculate the point-in-time when $frameRcvd_b$ occurs. One is by $transWindow$, as shown in equation 3.8. It shows the actual arriving time of the data frame.

$$frameRcvd_b = twStart + transWindow \tag{3.8}$$

The other way is by $dataRcvd_b$ and the reaction delay between $frameRcvd_b$ and $dataRcvd_b$, as shown in equation 3.9. The same as $sendDelay_a$, there is also such a time delay $rcvDelay_b$, during which the data frame is unpacked and transmitted to the rport of the function. $dataRcvd_b$ is a time-triggered event, it has its own scheduled time to occur, so equation 3.9 shows a deadline, which is the latest point-in-time when $frameRcvd_b$ must occur, otherwise $dataRcvd_b$ will miss its schedule.

$$frameRcvd_b = dataRcvd_b - rcvDelay_b \tag{3.9}$$

The value of $frameRcvd_b$ is different by using equation 3.8 and 3.9. This difference illustrates that there exists a 'gab', which provides a moving space for the transmitting window. The earliest possible point-in-time of data frame transmitting is $frameReady_a$, the latest point-in-time when the transmission is finished is $frameRcvd_b$ in equation 3.9. This moving space allows the transmitting windows which occurs at the same time or partly overlapped on the time axis to be moved, so the collision is avoided.
The stimulus of $reactDelay_{app}$ is still $dataRcvd_a$ and the response is $dataSent_b$, the same as intra-ECU communication. There are also the event $frameRcvd_a$ on ECU_1, which occurs before $dataRcvd_a$ and $frameReady_b$ on ECU_2, which occurs after $dataSent_b$. These two events are not included in the $reactDelay_{app}$. It is also correct to set the stimulus of $reactDelay_{app}$ to $frameRcvd_a$ and the response to $frameReady_b$. This change won't affect the behavior of network media access.
The send delay and the receive delay between the flow port and the communication driver are depending on ECUs. The value of the delay is a worst-case estimation, which is a fixed value, so it can be considered the send delay and receive delay of the same ECU have the same value. Hence a so-called ECU delay is created, when

a function is allocated to an ECU, its $rcvDelay$ and $sendDelay$ will be set to this ECU delay, as shown in equation 3.10

$$sendDelay_a = rcvDelay_a = ecuDelay_1 \tag{3.10a}$$
$$sendDelay_b = rcvDelay_b = ecuDelay_2 \tag{3.10b}$$

Figure 3.1 and 3.3 only describe the situation that a function sends out data to and receives data from a single function. If a function receives data from multiple functions, there must be a event for each data-receiving action, the receiving period of each event should be defined. For the data sending, if the data is sent to multiple functions with different size and period, there must be a event for each data sending action too.

3.2. Scheduling Algorithm for Media Access

In an automotive system, the network nodes are ECUs. The software components of an ECU generates data and the ECU put the data frame on the network. It can be considered when and how much data is transmitted from an ECU is decided by its software components. In the real-time network, Besides time-triggered (TT) traffic, there is also event-triggered traffic in the network, including rate-constrained (RC) traffic and best-effort (BE) traffic, which have lower priority than TT traffic. When TT traffic takes all the time slots, other traffics can not be transmitted. Each software components has its own fixed data-receiving deadline for TT traffic, and the data-sending is time-triggered, i.e. the data is sent periodically. In order to meet the deadline and avoid the transmission collision among network nodes, the network scheduler must accurately allocate time slots to each network node.
The scheduling algorithm introduced in this section is based on Earliest Deadline First (EDF) scheduling. The data-receiving deadline of a function is the $frameRcvd$ defined in Section 3.1. The scheduler must insure the data is received before this point-in-time.

3.2.1. The Concept of Scheduling Axis

In Section 3.1 it is discussed that it is necessary to know some conditions in order to schedule the network media access, including the point-in-time when the application and each function is triggered, the reaction delay of the application and each function. The $startOffset$ is used to indicate when a function starts. The $startOffset$ of the first function in an application is also used as the triggered time of the application. With this start point and the reaction delay, when the data is sent out by the function can be calculated, hence when data is transmitted on the network can be scheduled.
The occupying time duration of the network media is the $transWindow$. A so-called *scheduling axis* is defined. It is a time axis, on which the $transWindow$s of all the

functions within a complete communication cycle are arranged. The communication cycle is calculated by equation (3.7). It is a LCM of function's data-sending periods. If all the functions have the identical period, then in a communication cycle, the $transWindow$ of a function will repeat only once. Otherwise, if they have different periods, $transWindow$ will repeat multiple times. Based on the scheduling algorithm, the repeated $transWindow$s are allocated along the *scheduling axis*.
An example is illustrated in figure 3.5 to explain the *scheduling axis*. This example will be used again as the use-case for the evaluation in Chapter 5. It is a simple system, which contains only two applications. Each application is composed of four atomic software components. It is assumed one software component is mapped to one EAST-ADL function, hence in the use-case these two phrases are exchangeable. *swc_a* to *swc_d* are connected together, they belong to *Application_1*. And *swc_e* to *swc_h* belong to *Application_2*.

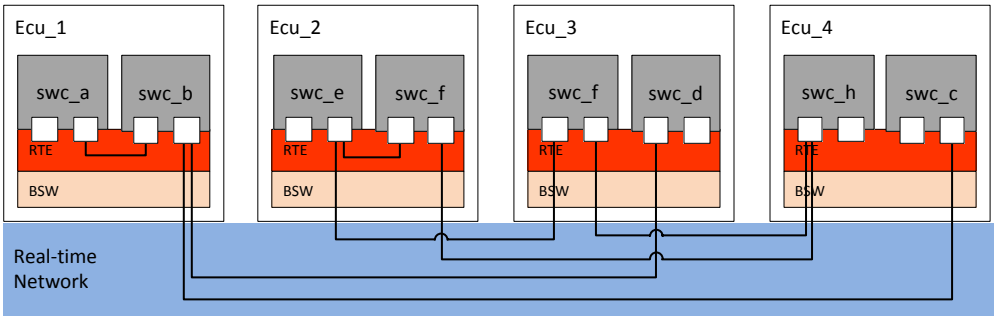

Figure 3.5.: A System Composed of Two Applications

There are totally seven connections between software components, five of them are inter-ECU communications, $b2c$, $b2d$, $e2g$, $f2h$ and $g2h$. The data-sending period on each communication is decided by the data-sending software component. If $sendPeriod\_b = 5$, $sendPeriod\_e = 10$, $sendPeriod\_f = 30$, $sendPeriod\_g = 20$, Then the $cycleLength$ is $LCM(5, 10, 20, 30) = 60$. The time unit used here is *time slot*, it is the smallest time block that can be allocated to network nodes in a real-time network. This simplifies the time of different granularity and makes it easier to illustrate in figures. So $sendPeriod\_b = 5$ means *swc_b* sends out data every five time slots. And the $cycleLength$ is 60 means there are sixty time slots in this cycle. The length of all the $transWindow$s is set to $1slot$.
The *scheduling axis* of the system is shown in figure 3.6. The colored blocks are transmitting windows of the inter-ECU communications, the gabs between them are idle time slots.
When a system is running, it can be assumed that a *scheduling axis* already exists, which is the original *scheduling axis* to be rescheduled during the dynamic network configuration. If new inter-ECU communications emerge, then the idle time slots in the *scheduling axis* will be allocated to them. The rescheduling is done by the scheduling algorithm, which will be introduced in the next two subsections. The

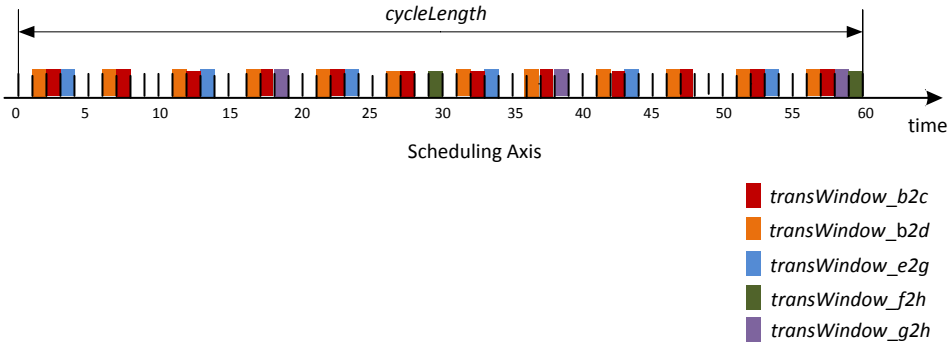

Figure 3.6.: Scheduling Axis of the System

example system in this section will be used to explain the algorithm.

3.2.2. Local Scheduling Algorithm

In figure 3.5, if Ecu is down, and swc_e is moved to Ecu_1 and swc_f is moved to Ecu_3, then connection $e2f$ becomes a new inter-ECU communication, the scheduler will look for idle time slots which are large enough to fit in all the repeated $transWindow_e2f$s. The searching range can be either 'local' or 'global'. 'local' means all the $transWindow$s of other communications are not 'moved' to make room for the new $transWindow_e2f$s. The scheduling only concerns the new windows, hence it is called local scheduling algorithm. 'move' means change the $twOffset$ of a $transWindow$, as it moves along the *scheduling axis*. 'global' means the existing $transWindow$s can be moved, the scheduling affect transmitting windows of multiple software components, in the extreme situation it may reschedule all the transmitting windows, hence it is called global scheduling algorithm. The advantage of local scheduling is it is simpler and quicker than global scheduling, the scheduled $transWindow$s won't be rescheduled. It is applicable when the network load is not heavy, because there will be enough idle time slots for the new $transWindow$s. When the network load becomes heavy, there are not enough idle time slots to fit in every repeats of the new $transWindow$, but they can be fit in if some scheduled $transWindow$s moves a little bit, then this is the time to apply the global scheduling. In this subsection the local scheduling algorithm is introduced, which is based on the EDF scheduling.
There is a searching range to search for time slots for $transWindow\_e2f$ on the *scheduling axis*, as shown in figure 3.7.
The new inter-ECU communication emerges because a software component is moved from one ECU to another. The point-in-time of the software component moving is named $pitMove$. The occurring time of software component moving is unpredictable. Because the data-sending period of $e2f$ is decided by swc_e, which is not a new period value, so the length of the communication cycle is not changed. $sendPeriod\_e$ is 10, so there are six $transWindow_e2f$s in a communication cycle.

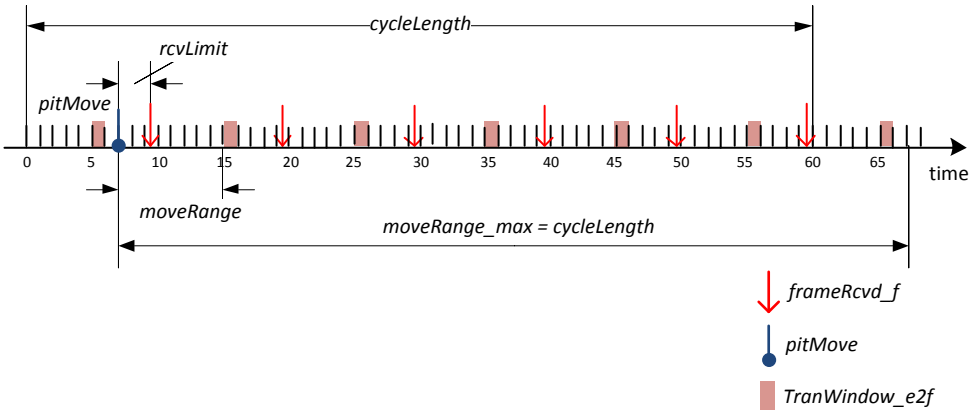

Figure 3.7.: Searching Range on A Scheduling Axis

The $transWindow\_e2fs$ are moved along the *scheduling axis*. The moved distance is named $moveRange$, which is from $pitMove$ to the point in time when all the $transWindow\_e2fs$ are fit in the axis. The $moveRange\_max$ is equal to $cycleLength$, because if the $transWindow\_e2fs$ have been moved across the whole $cycleLength$, it will reach the point-in-time in the next cycle when the $pitMove$ has occurred.

$$moveRange = twStart\_e2f\_1 - pitMove \tag{3.11}$$

$$moveRange\_max = cycleLength \tag{3.12}$$

The data-receiving period of swc_f is equal to of $sendPeriod\_e$. So in the communication cycle there are six $frameRcvd\_fs$, which are the arriving deadlines of the $transWindow\_e2fs$. $rcvLimit$ is the time duration from $pitMove$ to the first $frameRcvd\_f$.

$$rcvLimit = frameRcvd\_e2f\_1 - pitMove \tag{3.13}$$

If within the $rcvLimit$ the data can be sent from swc_e to swc_f, then no deadline is missed. It is quite possible that within the $rcvLimit$ there is no fitted time slot for a $transWindow\_e2f$, the $transWindow\_e2fs$ must be moved on, so one deadline is missed, figure 3.7 illustrates such a situation. If no time slots are found until the next $frameRcvd\_f$, another deadline is missed. How many deadlines a transmitting window can miss depends on the tolerance of the receiving software component. If the $transWindow\_e2fs$ have been moved across the whole communication cycle, and no fitted time slots are found, then global scheduling is applied.
The new $transWindows$ are put in the *scheduling axis* as the following steps:

(1) All the communications are sorted according to the data-receiving deadline of the data-receiving software components.

(2) The length of the communication cycle $cycleLength$ is calculated by equation (3.1).

(3) For the new inter-ECU communication $e2f$, its data-sending software component is swc_e. swc_e's data-sending period is $sendPeriod\_e$, the number the $transWindow\_e2f$ repeats in one communication cycle is $twNum\_e2f$:

$$twNum\_e2f = \frac{cycleLength}{sendPeriod\_e} \tag{3.14}$$

(4) For swc_e, the $startOffset\_e$ and $reactionDelay\_e$ are given conditions. The transmitting offset of the first $transWindow\_e2f$ is $twOffset\_e2f\_1$. If the $twOffset\_e2f$ of each repeated $transWindow\_e2f$ is set to 0 as the initial value, then the start position of the first $transWindow\_e2f$ on the *scheduling axis* is:

$$twStart\_e2f\_1 = startOffset\_e + reactionDelay\_e + 0 \tag{3.15}$$

(5) The position of the n_{th} transmitting window $transWindow\_e2f$ on the *scheduling axis* is:

$$\begin{aligned} twStart\_e2f\_n =& startOffset\_e + reactionDelay\_e + 0 \\ &+ (n-1) * sendPeriod_e \\ (0 \leq& n \leq twNum\_e2f) \end{aligned} \tag{3.16}$$

(6) The length of the $transWindow\_e2f$ is calculated by equation (3.7).

(7) $transWindow_e2f$s are moved along the *scheduling axis*. They are compared to other $transWindow$s. If they overlap with other windows, the transmitting offset will be reset. For example, if $transWindow\_e2f\_n$ overlaps with the m_{th} $transWindow\_b2c$, the $twOffset\_e2f\_n$ is reset like this:

$$\begin{aligned} offsetDiff\_e2g\_n =& \mid twStart\_b2c\_m + transWindow\_b2c \\ &- twStart\_e2g\_n \mid \\ twOffset\_e2g\_n =& offsetDiff\_e2g\_n \\ (0 \leq n \leq& twNum\_e2f, 0 \leq m \leq twNum\_b2c) \end{aligned} \tag{3.17}$$

The $transWindow_e2f$s are fitted by the local scheduling on the *scheduling axis*, as shown in figure 3.8.

3.2.3. Global Scheduling Algorithm

If the new emerged $transWindow$s have been moved across the whole communication cycle, and no fitted time slots are found, then global scheduling is applied. It moves other scheduled $transWindow$s to make room for the new ones. The global scheduling process is shown in figure 3.9.
In the example, $sendPeriod\_e$ is changed from 10 time slots to 20 and $transWindow\_e2f$ becomes 2 time slots instead of 1. At first, by using the local scheduling, fitted time

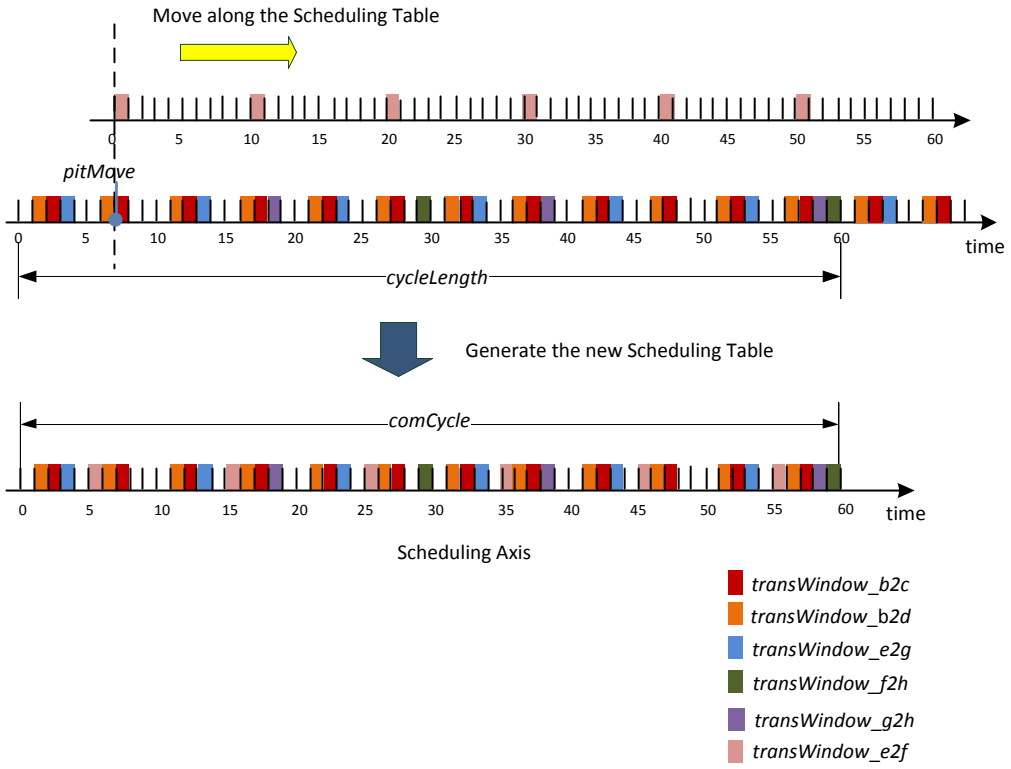

Figure 3.8.: Local Scheduling on Scheduling Axis

slots are found for the $transWindow_e2f$s, the $moveRange$ is from $pitMove$ at 15 to 33. But swc_f can not accept such a delay. In the figure it is illustrated that if $transWindow\_e2f\_1$ is moved to 19, then except one time slot occupied by $transWindow\_f2h_2$ at 60, other $transWindow_e2f$s can be fitted into the scheduling axis, and the $moveRange$ is 4, such a delay is acceptable by swc_f. So moving $transWindow_f2h$s is a possible solution.
When $transWindow_f2h$s are moved, there will be two situations. One is there are enough idle time slots for the $transWindow_f2h$s, so no more scheduled transmitting windows are affected. The other situation is moving $transWindow_f2h$s causes more scheduled transmitting windows to be moved. In the figure, the first situation is illustrated, all the $transWindow_f2h$s are moved 4 time slots to the left without disturbing other scheduled windows and all the $transWindow_e2f$s are fitted into the scheduling axis.
The global scheduling can be considered as an iteration of local scheduling. When $transWindow_f2h$s are moved, the steps in Subsection 3.2.2 is implemented.

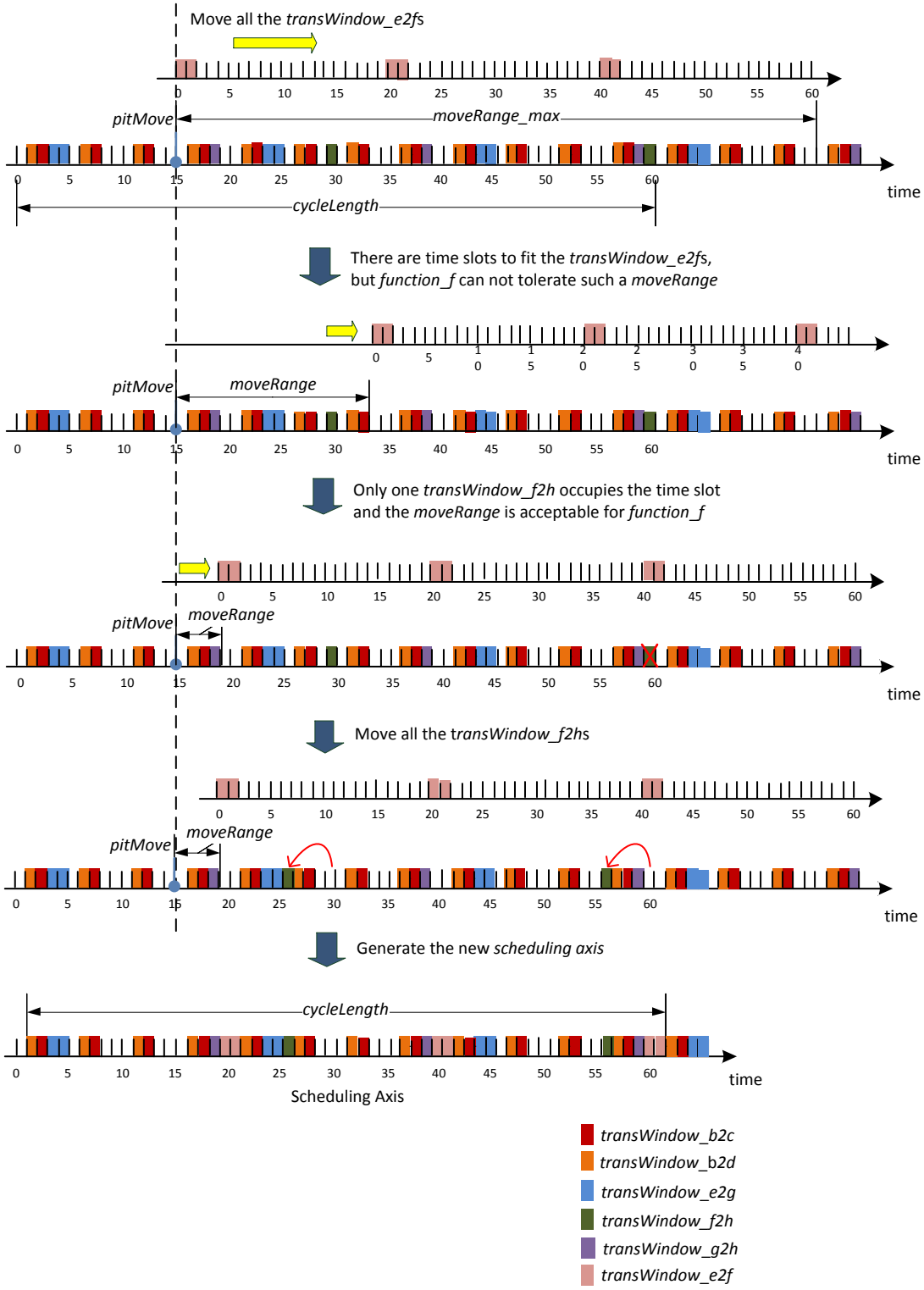

Figure 3.9.: Global Scheduling on Scheduling Axis

4. Implementation of Communication Model and Plugin for Model Transformation

In order to implement the function communication modeling concept in an AUTOSAR system, the communication model should be described in a standard modeling language which conforms to AUTOSAR. EAST-ADL [36] is a modeling language which is suitable to model an automotive system.
In this chapter first EAST-ADL and its relation with AUTOSAR are introduced. Then the communication behavior between functions is implemented in EAST-ADL modeling. And a model-to-text transformation plugin is created. The scheduling algorithm defined in Chapter 3 is realized in the plugin and a simulation tool is integrated. The plugin provides a solution to validate the network performance of an automotive system in the early phase.
The transformation plugin is written in Java and Xtend.

4.1. EAST-ADL Modeling Language

EAST-ADL is short for Electronics Architecture and Software Technology - Architecture Description Language. It is a architecture description language which provides a system modeling architectural framework for organizing and representing engineering information. It is used for capturing 'automotive electrical and electronic systems with sufficient detail to allow modeling for documentation, design, analysis, and synthesis' [7].

4.1.1. Model Structure

An EAST-ADL model is organized in several abstraction layers, which is shown in figure 4.1. Each level is a complete representation of a vehicle embedded system and it has a clear separation of concerns from other levels. From the top level to the bottom level the functionality becomes more and more concrete.
The *Vehicle Level* is the most abstract level. It characterize a vehicle by means of features, including functional and non-functional characteristics. The features are specified by requirements and use cases.
The next level is *Analysis Level*, which realizes functionality based on the requirements. It represent the abstract functionality of the E/E system without imple-

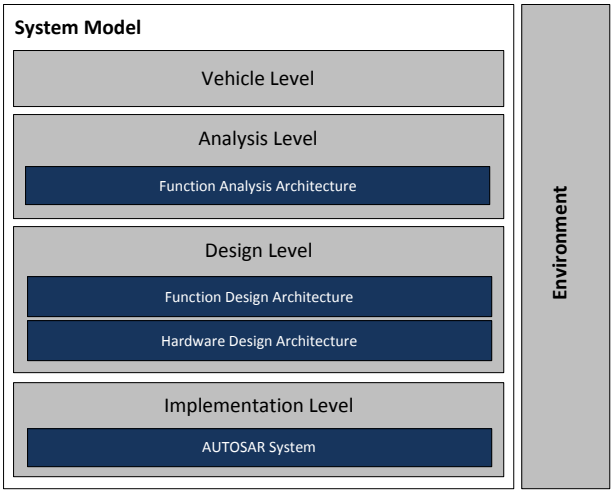

Figure 4.1.: EAST-ADL Model Structure [7]

mentation details. This level includes the *Functional Analysis Architecture* (FAA), which analyzes the vehicle feature and maps the feature to function entities.
Design Level defines the functional definition of software, the functional abstraction of hardware and middle-ware and the the function-to-hardware allocation for the implementation. This level includes the *Function Design Architecture* (FDA) and the *Hardware Design Architecture* (HDA). The FDA represents a decomposition of functionality in the FAA and maps the FAA entities to FDA entities. The HDA maps the functional devices in the FAA to the HDA entities. It models ECUs, communication links, sensors and actuators, and their connections.
Implementation Level represents the software implementation of the system. The FDA entities are mapped to software components. Hardware are represented by ECU specifications and topology. The model in is captured Software Component Template, ECU Resource Template and System Template in AUTOSAR. Environment contains environment functions, which are models of the vehicle behavior and the non-electronic systems. Environment is needed for validation and verification.
The features of an automotive system are designed on the vehicle level. On the analysis the features are mapped to functions, which are mapped to a bunch of functions and then allocated to hardware on the design level. The network access scheduling needs information of the function-to-hardware allocation. The functionality of the system does not concern the scheduling method. So there is no need to model the system on the vehicle level. The analysis level does not contain information of function-to-hardware allocation too. Hence in this thesis the use-case is modeled directly on the design level.

4.1.2. EAST-ADL to AUTOSAR Mapping Relations

The software architecture is defined by the software engineering. EAST-ADL complements AUTOSAR with higher abstraction levels and wider scope. EAST-ADL

supports the software engineering for automotive embedded systems and AUTOSAR captures the software architecture. The functional structure and behavior, the hardware topology and the function-to-hardware allocation on the design level of EAST-ADL are constraints for AUTOSAR software architecture.
The functional architecture and the software architecture is orthogonal. The functional architecture is a system decomposition which defines the logical parts of the system and how they interact. The software architecture is a system decomposition which defines the final product of an implementation. The same functional architecture can be mapped to different software architectures. The structural compliance between the two architectures are shown in figure 4.2.

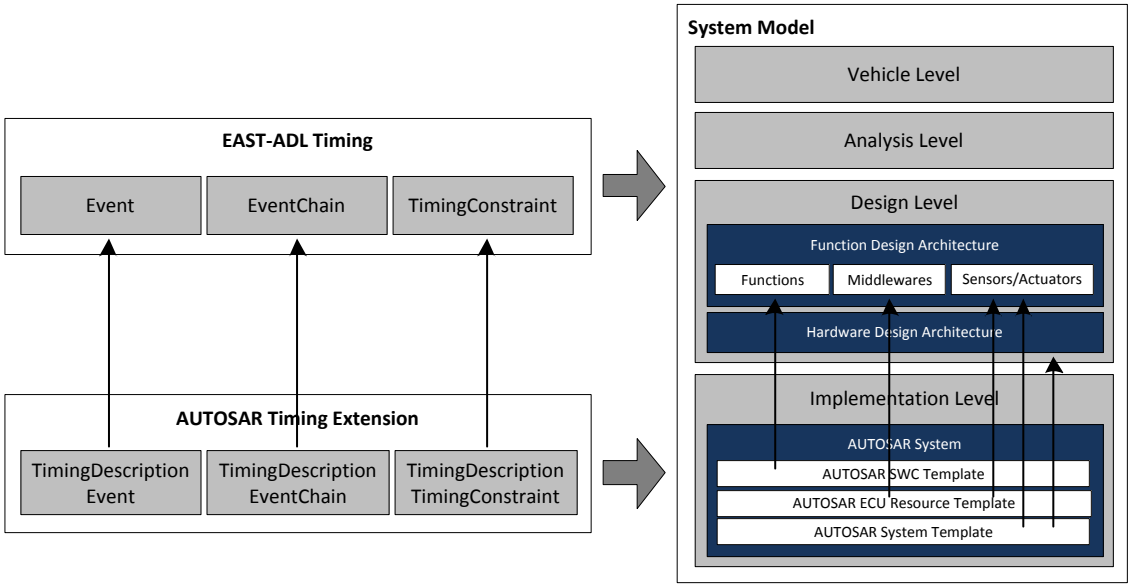

Figure 4.2.: Structural Compliance of Functional Architecture and Software Architecture [8]

DesignFunction represents structure and behavior in EAST-ADL and runnable is the basic behavior entity in AUTOSAR. *DesignFunction* to software component mapping can be various. One or multiple *DesignFunction*s can be mapped to one runnable. The mapping needs precise behaviorally and structurally relation between *DesignFunction* and the runnable. *DesignFunction* can be also mapped to one or more atomic software components. This mapping treats the software component as black box. *DesignFunction* can also be mapped to software compositions. This mapping treats the composition as black box.
Events in EAST-ADL denote the state change in a *DesignFunction*. And events in AUTOSAR denotes the state change in a running system on different views, not all the events in AUTOSAR have corresponding mapping events in EAST-ADL, such as the *AUTOSAREvent*, which refers to any event defined in AUTOSAR that does not have a corresponding event in EAST-ADL.

4.2. Implementation of Function Communication Model in East-ADL

On the design level of EAST-ADL, an automotive system is composed of functional design architecture (FDA) and hardware design architecture (HDA). FDA and HDA reflect the system architecture, the behavior of the system is described by timing elements. In order to implement the function communication model in the system, the timing elements are added in the EAST-ADL model and related to HDA and FDA.
EAST-ADL provides various types of elements. The hardware elements are used to create hardware architecture. In an automotive network, besides ECUs there are also sensors and actuators. The data transmission between sensor/actuator and ECU not via bus but direct, i.e. sensor/actuator won't need to access the network. Hence in this thesis there are only ECUs created the network, sensors/actuators are omitted.
The hardware element *Node* represents a computer node of the embedded E/E system, it is used to define an ECU type. In EAST-ADL when a hardware is created, it is actually a hardware type, the elements added in the HDA are instances of the created hardware type, not the hardware type itself. And a hardware type can have multiple instances. A hardware instance is represented by *HardwareComponentPrototype* and it can be used to create an ECU instance.
A hardware type has hardware pins for connection to the system, such as the power pins or communication pins. *CommunicationHardwarePin* represents the hardware connection point of a communication bus. A *Node* with *CommunicationHardwarePin* is able to connect to the network. The ECU instances of the same type have identical *CommunicationHardwarePin*s. In HDA all the ECU instances are connected to network by *HardwareConnector*s through *CommunicationHardwarePin*s. *HardwareConnector* represents electrical wire that connects hardware components. After the physical connections are set up, *HardwarePortConnector* is created in HDA, which defines the properties of the network, such as the bus type and transmission speed. The *HardwareConnector*s which belong to the same *HardwarePortConnector* compose a network which conform to the same protocol.
The functional elements are used to create functional architecture. The same as creating hardware, in EAST-ADL when a function is created, it is actually a function type, and a function type can have multiple instances. The elements in FDA are instances of the created function type. In an automotive system there are different function types, such as *DesignFunctionType*, *BasicSoftwareFunction*, *HardwareFunction* or *LocalDeviceManager*. *DesignFunctionType* is for modeling the functional structure, which can be mapped to the software component on the application layer in AUTOSAR. The type of a function won't affect the network media access, so it is not necessary to create different function types in the system, only *DesignFunctionType* is implemented. A function type may have multiple instances. A function instance is represented by *DesignFunctionPrototype*.

A function type has ports for connection to other functions. There are two kinds of ports, *FunctionFlowPort* is used for sender-receiver communication. And *FunctionClientServerPort* is used for client-server communication. As the type of communication won't affect the network media access, only *FunctionFlowPort* is implemented in the system. In FDA function instances are connected by *FunctionConnector*s through *FunctionFlowPort*s. Function communication network is decided by function connectors.
Function connector is different from the network communication. Function connector only shows the data flow between functions, the data flow can be either via RTE or via the network, which is decided by the function-to-hardware allocation. If the connected functions are allocated to the same ECU, the data flow can be considered as intra-ECU communication, if they are allocated to different ECUs, the data flow is considered as inter-ECU communication. *FunctionAllocation* represents the allocation. It binds a function prototype on a hardware prototype. *FunctionAllocation* sets up the relations between FDA and HDA and it is an essential element to decide the network access scheduling.
In order to implement the communication modeling concept in an AUTOSAR system, the events and constraints defined in function communication model is mapped to EAST-ADL timing elements. The mapping relations are shown in figure 4.3.

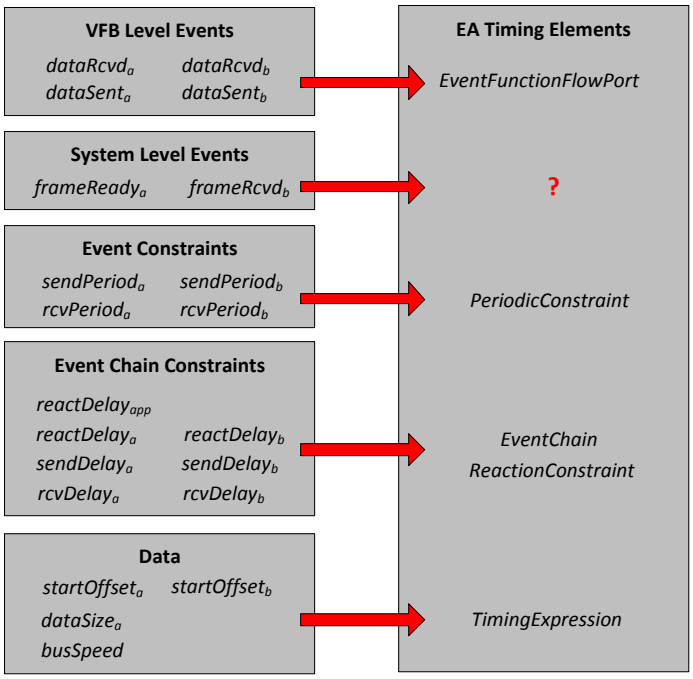

Figure 4.3.: Mapping Relations to EAST-ADL Timing Elements

According to AUTOSAR timing extension, the events and constraints of the communication model can be sorted into different types. The events on the function's flow ports on the application layer, so they are VFB level events, the correspond-

ing timing event type in EAST-ADL is *EventFunctionFlowPort*. The events on the ECU's communication driver are events on the BSW layer, so they are system level events, such events only exist on the implementation level of EAST-ADL, i.e. in AUTOSAR. On the design level there is no corresponding timing elements to express them. But with equation (3.5), (3.8) and (3.9) the point-in-time when this type of events occur can be calculated. The event constraints limit the point-in-time and the frequency of the event occurrence. The events time-triggered and periodic. The corresponding timing constraint type in EAST-ADL is *PeriodicConstraint*. But this element only defines the occurring period of the event, it does not define when it occurs, i.e. the offset of an even from the cycle start. *TimingExpression* can be used to define the data value, the start offset will be defined by *TimingExpression*.
The reaction delay of application and function are constraints of event chains. The corresponding EAST-ADL timing elements to express the delay is *ReactionConstraint*, in which the delay value is defined. The event chain is expressed by *EventChain*. In the *EventChain* the stimulus and response events are defined.
Besides the event, event chain and its corresponding constraint, there are also some data types to be defined. Such as the network bus speed and the size of the data. The corresponding element in EAST-ADL is *TimingExpression*.
After HDA and FDA are created, the system behavior can be defined. The function communication model is implemented by the timing events and constraints. Figure 4.3 illustrates the EAST-ADL timing elements implemented for the communication model. But the dependency of the timing elements to HDA and FDA is not set up yet, i.e. It does not indicate which event occurs in which part of the system.
EventFunctionFlowPort occurs on the *FunctionFlowPort* of a function, *PeriodicConstraint* constrains the behavior of *EventFunctionFlowPort*. *EventChain* is a chain of *EventFunctionFlowPort*s. *ReactionConstraint* constrains the behavior of *EventChain*. It is clear that *EventFunctionFlowPort* is a key element to set up relations of all the elements. If the dependency between *EventFunctionFlowPort* and *FunctionFlowPort* is set up, then all the timing events and constraints on the VFB level and the system are related together. The offset from the cycle start to the point-in-time when the function is triggered is also an essential timing constraint. It indicates when a function should start. The value of the start offset is represented by *TimingExpression*, which should be related to *FunctionFlowPort*.
The events on the system level are not available in EAST-ADL, but the point-in-time when the events occur can be calculated based on the reaction delay of ECU. The value of the delay is a property of ECU and it is represented by *TimingExpression*. The delay value is related to the ECU instance *HardwareComponentPrototype*.
Other data such as bus speed and data size are represented also by *TimingExpression*. Bus speed is related to the network, so it should be related to *HardwarePortConnector*. The transmitted data is generated by function, it is a property of function prototype, so it should be related to a function instance *DesignFunctionPrototype*.
How the relations are set up is open. There are multiple ways to realize that. It also depends on which EAST-ADL modeling tool is utilized. In Chapter 5 an use-case

is created, how to set up the relation between the timing elements and the system is illustrated.

4.3. Model Transformation Plugin and Simulation Tool Integration

With the implementation of the communication model in EAST-ADL, an automotive system is able to support dynamic network configuration. The configuration is realized by the scheduling algorithm developed in section 3.2 of Chapter 3. But EAST-ADL tools can not provide such an environment in which the network can be configured by the scheduling algorithm, so a simulation tool is needed. The EAST-ADL model is transformed to simulation files, during the transformation, the scheduling algorithm is implemented to set up parameters for the simulation. In this way, the communication model and the scheduling algorithm can be evaluated. A model transformation tool is needed to integrate the simulation environment, which provides a solution to validate the system design in an early phase. OMNeT++ [37] is a simulation library and framework for building network simulators, which provides real-time network plugins such as TTEthernet and AVB. It can be used to evaluate the performance of the network. An Eclipse plugin is created as the model transformation tool, in which the scheduling algorithm is realized and the EAST-ADL model is transformed into OMNet++ text files.
In OMNeT++ there are two kinds of simulation texts: the ini files and the ned files. The ned files defines the network nodes and the network architecture. The ini files are used to configure the network nodes and the network. The information of the EAST-ADL model is extracted and mapped into the ini files and the ned files. The information needed by a specific ned or ini file is from different part of the EAST-ADL model. To set all the parameters in the ned or ini file, the model must be checked through for many times. It increases the complexity of the searching algorithm and makes the mapping relations unclear. Also the code becomes difficult to maintain. One way to make the work easier is to define a middle model, the model is created based on the content of the ned and ini files. The EAST-ADL model is first transformed to this middle model, and then the middle model is transformed to simulation files. The transformation method is shown in figure 4.4.
First related information are extracted from the EAST-ADL model and saved in the Ecore model by model-to-model transformation. Then the ned and ini file are generated by the model-to-text transformation based on the Ecore model. The modeling tool Eclipse Sirius [38] is applied to create an Ecore model, which is applied as the middle model.

4.3.1. Ecore Model for Model-to-Text Transformation

The Ecore model is created based on the ned and ini files. Normally there is one ned file which defines the network architecture, the network nodes are included in it.

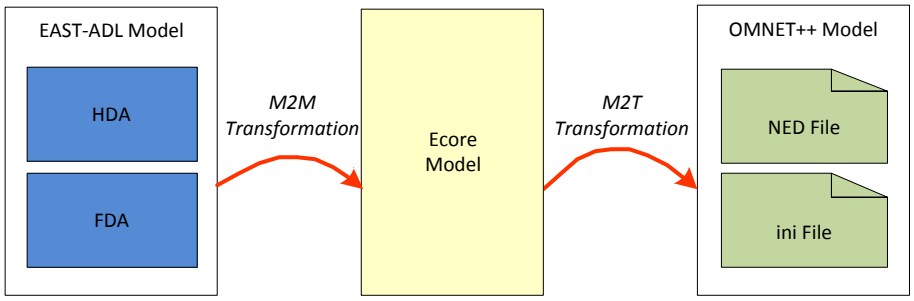

Figure 4.4.: Model Transformation Method

And the network type and topology is also defined. The network node in this ned file are actually instances of network node types. A network node type is defined in an independent ned file. And a network node type can have multiple instances. The network node type can be terminals such as ECU. And if the network is switch-based, there is also ned files to define a switch type.
For each ned file there is a corresponding ini file. For the ned file which defines a network node type, its corresponding ini file configures parameters such as the data transmitting point-in-time, transmitting period or the connection ports. For ned file of a switch type, the ini file defines the switch data flow inside the switch. There is also an ini file for the network architecture ned file, in which parameters such as simulation time length, scheduler, cycle length are configured. The Ecore model for OMNeT++ transformation is illustrated in figure 4.5, the yellow squares are classes, each class has a corresponding OMNeT++ element. The properties added in the class are the parameters of the elements.
To create a simulation scenario in OMNet++, first the network architecture should be defined. Class *Network* is mapped to the network architecture ned file. The network is switch-based, hence *Network* has a composition to the class *Switch*, it means the network is formed by multiple connected switches and all the network nodes are connected into the network via the switches. *Network* only has one property, the name of the network.
Class *Switch* is mapped to the switch type ned file. The instance of *Switch* will be declared in the network architecture ned file. The network speed is defined in switch's property *busSpeed*. *Switch* has a composition to the class *Ecu*, it means multiple ECUs are connected to the network through switch. A switch handles data transmission between network nodes. Hence *Switch* also has a composition to the class *TTIncoming* and *TTDoubleBuffer* respectively. *TTDoubleBuffer* and *TTIncoming* are defined by the CoRE4INET plug-in (CoRE4INET plug-in will be introduced in Chapter 5). *TTIncoming* is used for a network node to receive data either from a function or from its own physical port. *TTDoubleBuffer* is used for a network node to send data either to a function or to its physical port at the scheduled time. In *Switch* there are multiple pairs of *TTDoubleBuffer* and *TTIncoming*, each pair handles the data from one specific physical port and sends it to the port which

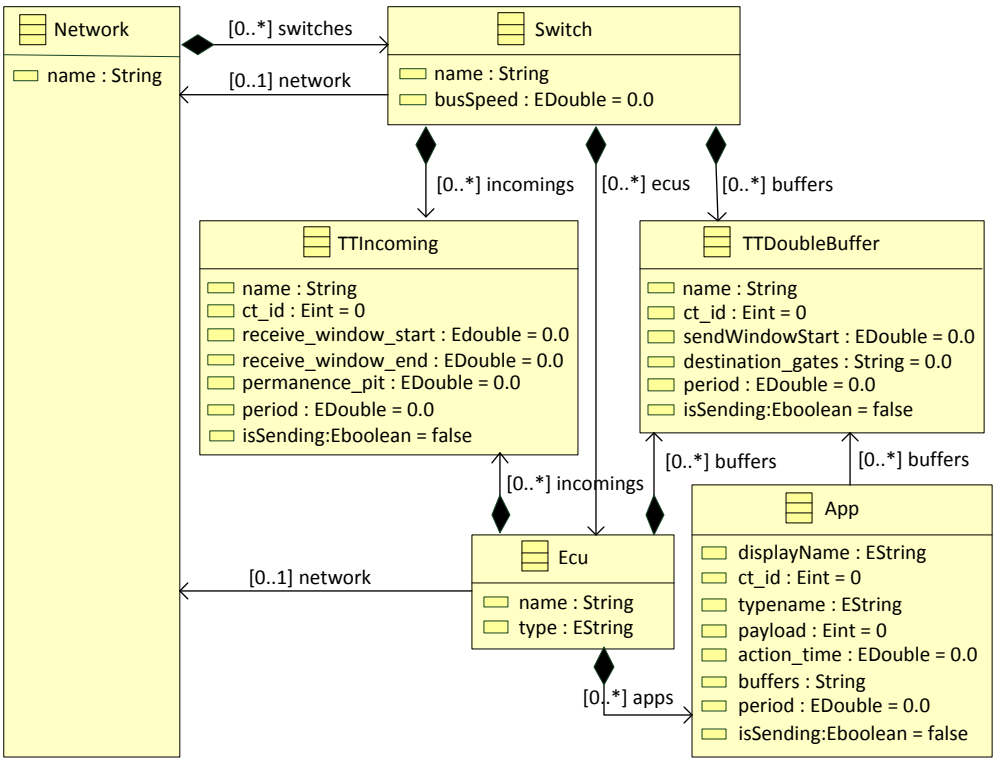

Figure 4.5.: The Structure of Ecore Model

connects the destined network nodes.
The network nodes are ECUs. Class *Ecu* is mapped to ECU type ned file. The instance of *Ecu* will be declared in the network architecture ned file. The property *type* is used to identify the ECU type. It is not a parameter in the ned file, it will be used for model transformation. *Ecu* has a composition to *App*, which means an ECU contains one or more applications which send out and receive data. The application in OMNeT++ is not an AUTOSAR-defined application, it is equal to a function or a software. *Ecu* also has a composition to class *TTIncoming* and *TTDoubleBuffer* respectively as *Switch*. In *Ecu* there can be multiple pairs of *TTDoubleBuffer* and *TTIncoming*, each pair handles a specific data traffic. The data traffic is either from the physical port of *Ecu* or generated by the *App* on the *Ecu* and it is sent either to the out port of of *Ecu* or to the *App* on the *Ecu*.
Class *TTIncoming* is the data source of a data traffic. Each data traffic has its own ID, the property *ct_id* is the traffic identification. 'ct' is short for critical traffic, which means the traffic is time-critical, including TT traffic and RC traffic. In this case only the TT traffic is added into the network traffic. The *TTIncoming* and *TTDoubleBuffer* with the same *ct_id* belong to the same traffic. For time-triggered traffic, there is a period of receiving data from the function or the physical port of the network node. This period is defined in property *period*. All the network nodes are synchronized, each node will open a time window to wait for the coming data

at a scheduled time. The data comes either from the network node's physical port or from the application of the network node. The concept of the receive window is defined in the TTEthernet specification [39]. The length of the window is flexible. The detail of receive window setting is described in [39]. *receive_window_start* and *receive_window_end* are the start and end of receiving window. In TTEthernet specification a specific concept is named 'permanence'. It is a point-in-time when the network node which receives the data adjusts the time label of the received data and its own time according to the global time. It is the time all the data frames are ordered according to their original sending order. The property *permanence_pit*is used to define this point-in-time. In OMNet++ this parameter can be set the same value as *receive_window_end*. *TTIncoming* receives data from the *App*. The type of *App* decides *TTIncoming* whether the receives data which will be sent out on the network or it is received from the network. Hence one more property *isSending* is added to indicate the type of the data traffic. It is used for model-to-text transformation.
Class *TTDoubleBuffer* sends out data in a data traffic. The same as *TTIncoming*, *ct_id* is the traffic identification. *period* is the period of sending out data to the function or the physical port of the network node. For a time-triggered data traffic, the data is sent in a scheduled point-in-time. There is a send window, which defines the earliest and latest point-in-time to send the data. *sendWindowStart* is the earliest point-in-time when the data is sent out by *TTDoubleBuffer*. If the data is sent on the network, before *sendWindowStart* the data is packed into the network data frame and waits in the out buffer of the network node's driver. The sending time is decided by the scheduler. If the data is sent to the function of the network node, the sending time can be any value once the data is ready as long as it is no later than the receiving time. There is also the parameter *sendWindowEnd* which is not implemented in this case. This parameter is used to limit the transmitting time. In a TTEthernet network, the size of the data frame is the same as the standard Ethernet frame. The network will cut the data into segments if the data is larger than the frame size. So it is unnecessary to limit the transmitting time manually when there is no specific purpose. If the data from the corresponding *TTIncoming* is received from the physical port, then the data is to be sent to a function on the network node. If the data is received from the function of the network node, then the destination is the physical port of the network node. *destination_gates* decides to which destination the data goes. The same as *TTIncoming*, *isSending* indicates the type of the traffic. It provides the destination information for *destination_gates* in the model-to-text transformation. If it is true, the destination is the physical port, otherwise the destination is the function.
The application in OMNeT++ is defined by class *App*. *App* can be either the source or the destination of data. If the application consumes data, only three properties are needed to define such an application. *displayName* is the name of the application. *typename* identifies the type of the application, to indicate whether it is a data generating application or a data consuming one. If it generates data, there will be a *TTIncoming* receiving its data. If it consumes data, there will

be a *TTDoubleBuffer* sends data to it. And *isSending* is used for model-to-text transformation, the property *typename* is set according to the value of *isSending*. If the application generates data, it has more properties. Only the application which generates data has the property *ct_id*. The *TTIncoming* which receives data from the application will be given the same *ct_id*, and the same id is given to its corresponding *TTDoubleBuffer*. *period* is the data sending period. *payload* is the size of the generated data. action_time is the parameter only for the application which generates data. It is the time the application generates data, which will be sent by the *TTDoubleBuffer* at *sendWindowStart*.
The application in OMNet++ is different from the concept of application in AUTOSAR. Application in AUTOSAR is a special kind of function that has complete functionality and satisfies certain user's requirement, such as ACC or BBW. From the functional aspect, the functionality of an application is composed of multiple functions. From the software aspect an application is composed of multiple software components. In OMNet++ an application is the same with an software component in AUTOSAR. In EAST-ADL if a function is mapped to a software component, then the function can be mapped to an application in OMNet++.
The ned files only defines the physical connections of the network, data traffics are defined in ini files by *TTIncoming* and *TTDoubleBuffer*. For every data traffic there is a pair of *TTIncoming* and *TTDoubleBuffer*. Take the system illustrated by figure 3.5 as an example. The data is transmitted from *swc_b* to *swc_c* through connection *b2c*. *swc_b* is allocated to *Ecu_1* and *swc_c* is allocated to *ecu_3*. In OMNet++, *Ecu_1* and *Ecu_3* are connected to network by a switch. The *TTIncoming* of *swc_b* receives data which is generated by an application of *swc_b*, and transmits it to *swc_b*'s *TTDoubleBuffer*. The *TTDoubleBuffer* will sends the data to the physical port of *Ecu_1*. *Ecu_1* sends data to switch. In the switch there is a pair of *TTIncoming* and *TTDoubleBuffer* which treats the data traffic from *swc_b* to *swc_c*. The *TTIncoming* receives data from the physical port of the switch, transmits it to the *TTDoubleBuffer*. Then the *TTDoubleBuffer* will sends the data to the physical port which connects to *Ecu_3*. The switch then sends data to *Ecu_3*. The *TTIncoming* of *swc_c* will receive the data from the physical port of *Ecu_3*, sends it to the *TTDoubleBuffer*. The *TTDoubleBuffer* sends data to *swc_c*.

4.3.2. Model-To-Model and Model-to-Text Transformation

EAST-ADL model is transformed to Ecore model by model-to-model transformation first, and Ecore model is transformed to text by model-to text transformation. In the model-to-model transformation, first all the needed EAST-ADL elements are found and saved in lists, then the information of the elements are exacted and resorted. For example, a function prototype list does not contain information such as data transmitting period, function reaction time, which are saved in other lists. Hence a class which is used to collect all the information of a function prototype should be defined in the plugin.
Class $ReactionConstraintInfo$ collects all the information of a reaction constraint.

There are two kinds of reactions, the function reaction, which is the reaction of a single function, and the application reaction, which is the reaction of a function chain. The content of a reaction constraint is the same regardless its type, in which contains the stimulus event, the response event, the point-in-time of the start and end of the reaction and its duration. So this class will be subsetted into the class which collects information for function prototype and the class which collects information for application respectively.
Class $FunctionPrototypeInfo$ collects information for a function prototype, such as the function flow ports, the reaction constraint, the data sending period and a $ReactionConstraintInfo$, in which saves the reaction constraint of the function prototype. Function $setFunctionPrototypeInfo$ is created to set all the value of $FunctionPrototypeInfo$. The reaction constraint is extracted from the reaction constraint list. If the stimulus event of the constraint is identical to the data receiving event and the response event is identical to the data sending event, then it is the reaction constraint of the function prototype. With the data receiving time and the reaction duration, the point-in-time when the data is sent out can be calculated. The period constraint of the function prototype is extracted from the period constraint list. If the corresponding event of the period constraint is the data sending event, then it is the period constraint of the function prototype. The reaction delay of the ecu prototype to which the function prototype is allocated will be used to calculate the frame receiving and sending time.
Class $ApplicationInfo$ saves all the information of an application, including the function prototype which triggers the application and a $ReactionConstraintInfo$, in which saves the reaction constraint of the application. Function $setApplicationInfo$ sets all the value of $ApplicationInfo$. First, the function prototype which triggers the application is found. The trigger contains elements which are the reaction constraints of the application. As the data which is sent out by the triggering function prototype may reach multiple destinations, so there can be multiple constraints, which are saved in a list. The event sends out data from the triggering function prototype and the event which receives the data in the destination function prototype are saved. Every function trigger and its information are saved in a table for the model transformation.
Class $FunctionConnectorInfo$ includes information for a function connector, such as the two function prototypes it connects, the timing information of the prototypes. Depending on the function-to-hardware allocation, every function connector can be a possible inter-ECU communication, hence $FunctionConnectorInfo$ contains information such as the offset of data frame transmitting time and the communication ID. The data frame transmitting duration is decided by the sending function prototype, which is already included in its information. Function $setFunctionConnectorInfo$ sets all the value of $FunctionConnectorInfo$. The connector connects two function prototypes, one sends out data, the other receives data. The data frame transmitting offset is not set by $setFunctionConnectorInfo$, but by a function called $setTransOffset$.
In order to set transmitting offset for all the inter-ECU communication, function

findInterEcuCommunication is created. It finds out the inter-ECU communications based on the function-to-hardware allocation. The allocations are checked. The client is the function prototype, the supplier is the ecu prototype. If there are two allocations, which clients are identical to the function prototypes of the connector, then the suppliers of the allocations are compared. If the two suppliers are different, that means the two function prototypes are allocated to different ecu prototypes, the function connector is an inter-ECU communication. Otherwise, if the suppliers are the same, it is an intra-ECU communication. Every function connector which is an intra-ECU communication is saved in a list, then the communications are sorted according to EDF.
After having distinguished inter-ECU communications and intra-ECU communications, function *setTransOffset* sets the transmitting offset of the data frames. This offset is from the the point-in-time when the data frame is ready to be sent out to the point-in-time when the data frame is actually sent out. Two communication are compared to check if there exists transmitting collision. The collision occurs when one transmission starts during another one is still transmitting frame, the offset makes the transmitting time start after another has finished the transmission. All the communications are checked to make sure there is no collision. After the offset is set, the deadline is checked. There are two kinds of deadlines. First deadline is the point-in-time when the data frame is received by the receiving function prototype, which must be larger than the end of the data transmission. If not, a warning will be given to the user to indicate the offset value is invalid. The other deadline is the end of the application reaction delay. If the receiving function is the destination function in the application, then it is checked if the application reaction delay is violated. If any of the two deadlines is violated, a warning message is illustrated.
After all the information needed are extracted and sorted, the Ecore model is created. The tricky part is the generation of *TTIncoming* and *TTDoubleBuffer* in class *ECU* and *Switch*. The number of *TTIncoming* and *TTDoubleBuffer* pairs of *ECU* depends on the number of applications in the ECU, and it also depends the number of inter-ECU communications from an ECU. Create class *TTIncoming* and *TTDoubleBuffer* is realized by checking through the communication list. When the data has reached the destination ECU, the data will be sent from the ECU physical port to the application. This delay is set to 10% of the transmitting window.
Then *TTIncoming* and *TTDoubleBuffer* of *Switch* are created. In the EAST-ADL model, there is no definition of the network topology. In order to conform to the switch-connected topology in OMNeT++, the transformation plugin defines the switch, which receives data from the ECU and transmits it further to the destination ECU. The receive window and transmitting time of the switch are set in the *TTIncoming* and *TTDoubleBuffer*. Different from the *TTDoubleBuffer* of *Ecu*, the *TTDoubleBuffer* of *Switch* must know to which physical port the data is sent. In *Ecu* there is only one physical port. The delay between the point-in-time when the switch has received the data and the point-in-time it transmits the data is 25% of the transmitting window. If this delay is too short, the switch won't transmit the data in the first cycle.

After the EAST-ADL model is transformed to the Ecore model, the simulation text can be generated based on the Ecore model. The Xtend template technique is used to generate the ned and ini files. For example, the following code is part of the template which generates the ned file of an ecu prototype:

```
module ecu.type extends TTEEtherHost
{
        submodules:
         FOR incoming : ecu.incomings
                incoming.name : TTIncoming;
          ENDFOR

         FOR buffer : ecu.buffers
                buffer.name : TTDoubleBuffer;
          ENDFOR

        connections:
         FOR incoming : ecu.incomings
                FOR buffer : ecu.buffers
                        IF incoming.ct_id == buffer.ct_id
                                incoming.name.out --> buffer.name.in;
                         ENDIF
                 ENDFOR
          ENDFOR
}
```

5. Evaluation of the Function Communication Model

In this chapter a use-case is created by Papyrus EAST-ADL [40]. The values of the model are set in order to set parameters of ned and ini files for simulation in OMNeT++. The simulation result is analyzed.
Papyrus EAST-ADL [40] is chose as the tool to model the use-case. Papyrus runs on the EMT and provides an integrated environment for editing EMF models and supports modeling languages such as SysML, EAST-ADL and MARTE. It also provides diagram editors for EMF-based modeling languages. Papyrus EAST-ADL is a component of the Papyrus, which supports EAST-ADL modeling. The version of Papyrus EAST-ADL is 1.0.1.
OMNeT++ is a simulation library and framework for building network simulators. It is based on Eclipse. OMNeT++ offers a graphical run-time environment and a host of other tools. INET Framework [41] is a communication networks simulation package for OMNeT++. It contains models for wired and wireless network protocols, such as TCP, IPv6, Ethernet, 802.11, MPLS and so on. CoRE4INET [42], created by the CoRE working group, is an extension to the INET-Framework. It is for the event-based simulation of real-time Ethernet. Now it supports TTEthernet and IEEE 802.1 AVB functionality. The version of OMNeT++ is 4.5.

5.1. Use-Case Model for Evaluation

The use-case is modeled based on figure 3.5. In Papyrus EAST-ADL a model project is created with applied EAST-ADL profile and system structure. The *EAPackage*s which are used to store elements on different levels are automatically created when the project is initialized. The package $SystemModel$ is such an automatically-created package, in which stores the structure elements of the system on different levels, $MyDesignLevel$ contains elements on the design level. All the elements in $SystemModel$ are automatically created, as shown in figure reffig:papyea-model-d. Additional elements can be added manually later.
The EAST-ADL profile *DesignLevel* is implemented to $MyDesignLevel$. This package will be used to find elements in the model-to-model transformation. There are two prototypes included in $MyDesignLevel$, fda and hda. fda is typed by $MyFunctionalDesignArchitecture$, which is the FDA created automatically when the project is initialized. Profile *DesignFunctionPrototype* is implemented. fda will be used by the model transformation plugin to find functional elements on the de-

sign level. *hda* is typed by *MyHardwareDesignArchitecture*, which is the HDA created automatically when the project is initialized. Profile *HardwareComponentPrototype* is implemented. *hda* will be used by the model transformation plugin to find hardware elements on the design level. *allocation* is used to store the allocation diagrams and profile *Allocation* is implemented. The function-to-hardware allocation links are drawn in the allocation diagram. Because of certain bugs in Papyrus EAST-ADL the links in the diagram can not be recognized. This will cause some troubles in the model transformation, which will be discussed later in this section.

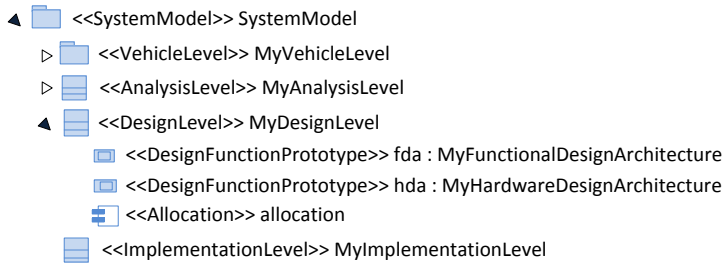

Figure 5.1.: System Model of Use-Case

In an EAST-ADL model project, there is only one object implemented with the profile *DesignLevel*. Hence $MyDesignLevel$ is a unique class in the model. Prototype fda is typed by the FDA in package $FunctionalElements$. Prototype fda is typed by the HDA in package $HardwareElements$. All the hardware and function instances are created in HDA and FDA, to which the hardware and function types are related. The timing elements are also related to FDA and HDA. Hence if the *hda* and fda in $MyDesignLevel$ are found, the HDA in $HardwareElements$ and the FDA in $FunctionalElements$ can be found, then all the hardware and function types and timing elements can be found. So $MyDesignLevel$ is an object, from which the transformation plugin starts to find all the needed elements to create the Ecore model. The *org.eclipse.papyrus.eastadl* plugin is added into the transformation plugin, which provides APIs to find EAST-ADL elements in the EAST-ADL model. As $MyDesignLevel$ is unique in the project, so profile *DesignLevel* is implemented only once in the project. *org.eclipse.papyrus.eastadl* provides the API to find profile of an object. So the plugin can find $MyDesignLevel$ by looking for the project with the profile *DesignLevel*.
All the hardware and function elements on the design level are stored in the package named $DesignLevelElements$. There are three more EAPackages in it to store different types of elements. The functional elements are stored in package $FunctionalElements$, the hardware elements are in package $HardwareElements$ and the timing elements are in package $TimingElements$. In the $HardwareElements$ the hardware types and HDA are defined, as shown in figures 5.2.

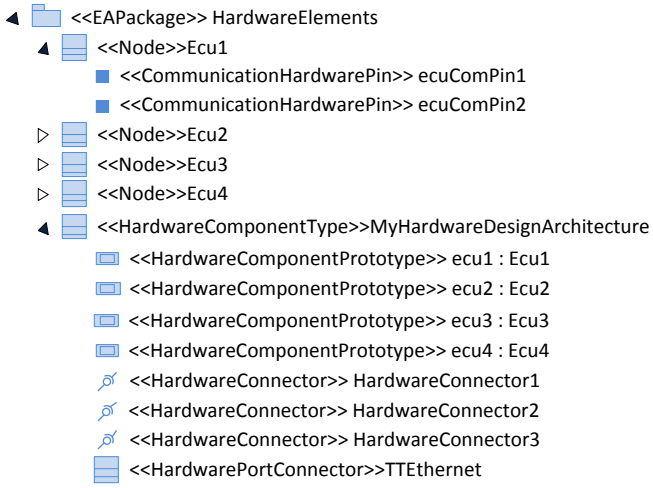

Figure 5.2.: Hardware Elements of Use-Case

Take *Ecu*1 for example, it is an ECU hardware type. The profile *Node* is implemented. *Ecu*1 has two hardware pins *ecu*1*ComPin*1 and *ecu*1*ComPin*2, to which the profile *CommunicationHardwarePin* is implemented. *ecuComPin*1 is created as an in port, so its property *direction* is set to *in*. *ecuComPin*2 is created as an out port, so its property *direction* is set to *out*. *MyHardwareDesignArchitecture* provides an empty object for creating HDA. It is created automatically when the project is initialized. A composite structure diagram named *hdadiagram* is added in HDA, which is used to draw the hardware architecture, as shown in figure 5.3. The elements drawn in the *hdadiagram* are automatically added in *MyHardwareDesignArchitecture*.

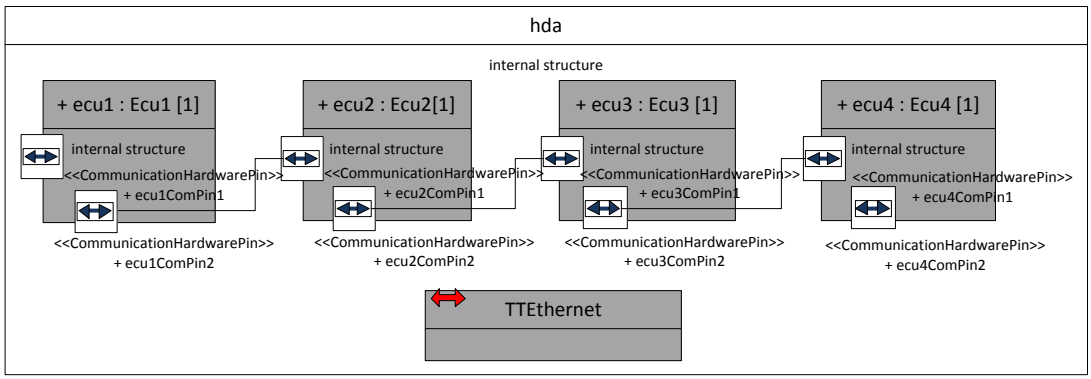

Figure 5.3.: HDA Diagram Of Use-Case

In *hdadiagram*, *HardwareConnector*1 connects hardware prototypes *ecu*1 and *ecu*2. The profile *HardwareConnector* is implemented. It does not mean that *ecu*1 and *ecu*2 are connected to each other directly. It only means these two prototypes are connected to the network through their *CommunicationHardwarePin*s. The topology of the network is arbitrary. The hardware connectors only connect all the ECU

instances with physical connections, the property of the network is not defined. The object named *TTEthernet* defines the network property. Profile *HardwarePortConnector* is implemented. The property *busType* is set to *TimeTriggered*, which means this is a time-triggered network of any possible protocol and topology. It can be a TTEthernet network, which is switch-based, or a TTCAN network, which topology is bus or a FlexRay network, which topology can be either bus, star or a mixture of both. The topology of the network won't affect the media access scheduling. Here it is considered the network conforms to TTEthernet specification. The property *busSpeed* is set to 100, the time unit is not defined, which will be set to *Mbps* in the model transformation plugin. The property *connector* is used to define the range of the network. The hardware connectors which are contained in this property belong to the same network. In the use-case all the three hardware connectors are included in *TTEthernet*. This means all the four ECU prototypes are connected into a time-triggered network.
The FDA and function types are defined in package *FunctionalElements*, as shown in figures 5.4.

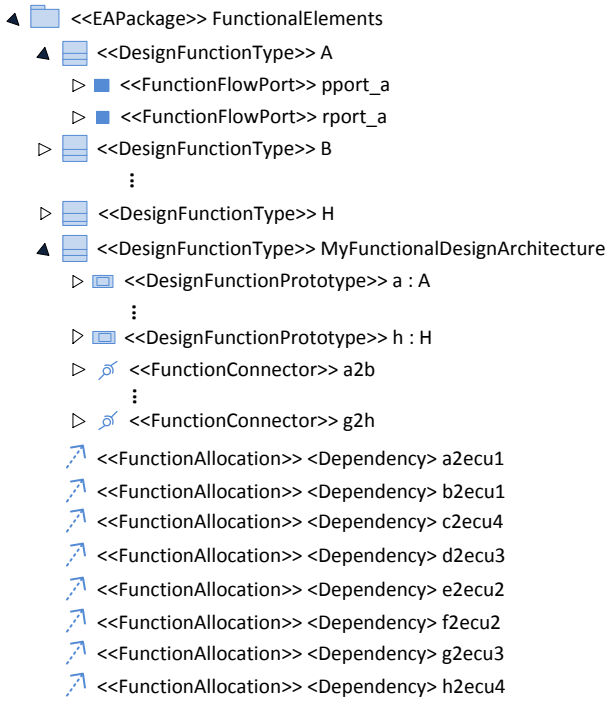

Figure 5.4.: Functional Elements of Use-Case

Take function type *A* for example. Profile *DesignFunctionType* is implemented. *A* has two function flow ports *pport_a* and *rport_a*, profile *FunctionFlowPort* is implemented. Flow ports are used in the sender-receiver communication. *pport_a* is created as a pport, so its property *direction* is set to *out*. *rport_a* is created as a rport, so its property *direction* is set to *in*. *MyFunctionalDesignArchitecture* provides an empty object for creating FDA. It is created automatically when the project

is initialized. A composite structure diagram *fdadiagram* is add in FDA, which is used to draw the functional architecture, as shown in figure 5.5. The elements drawn in the *fdadiagram* are automatically added in *MyFunctionalDesignArchitecture*.

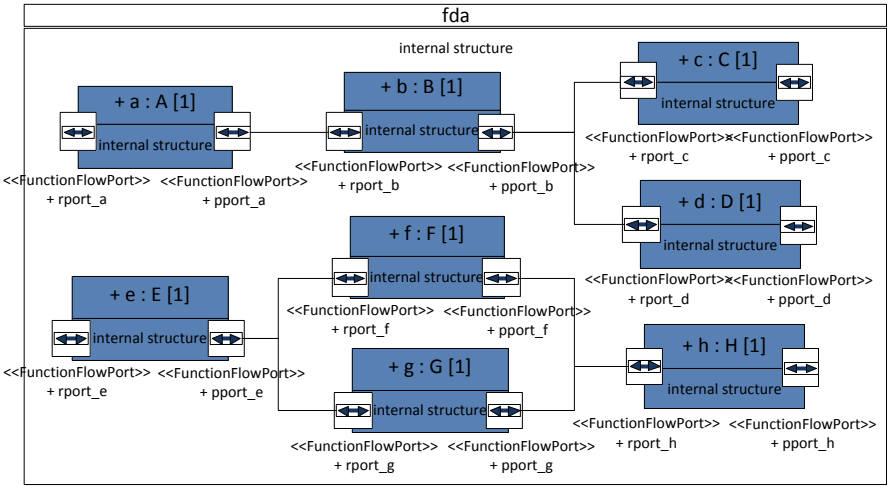

Figure 5.5.: FDA Diagram of Use-Case

A composite structure diagram *allocationdiagram* is created in *MyDesignLevel* to draw the function-to-hardware allocation links between FDA and HDA, as shown in figure 5.6. The allocation links drawn in the diagram are added in *FunctionalElements* automatically.

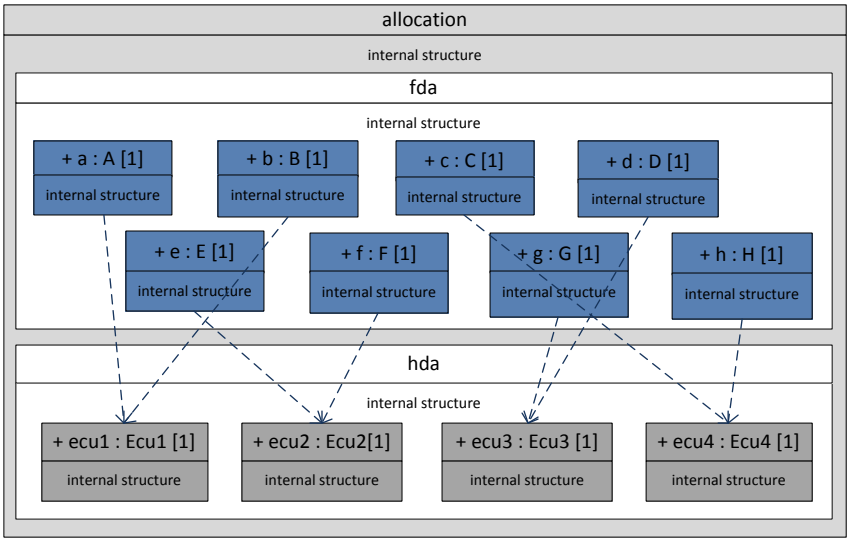

Figure 5.6.: Allocation Diagram before Software Reconfiguration

The function-to-hardware allocation in EAST-ADL is different from the software-to-ECU allocation in AUTOSAR. Function stands for a functionality part of the

system, it can be mapped to one or multiple software components. If a function is mapped to a atomic software component, then the function-to-hardware allocation can be mapped to the software-to-ECU allocation. Multiple allocation diagrams can be created in *MyDesignLevel*. Figure 5.6 illustrates the scenario before the software component is moved. Figure 5.7 illustrates one possible scenario after the software component is moved. In the figure, function prototypes in *ecu*2 are moved to other ECU prototypes.

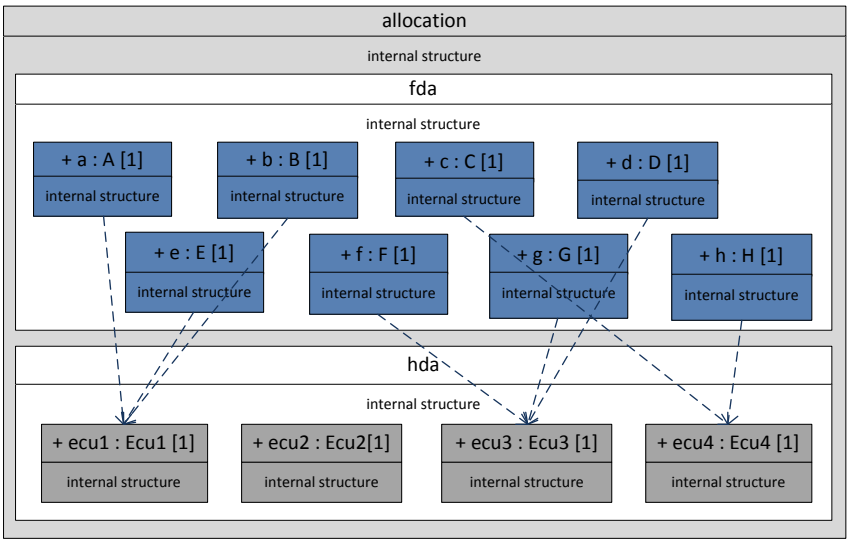

Figure 5.7.: Allocation Diagram after Software Reconfiguration

When *MyDesignLevel* is automatically created during the project initialization, the object implemented with the profile *Allocation* is also created, in which allocation diagrams can be added. The property *functionAllocation* can be used to identify allocation links belonging to different allocation scenarios. However because of certain bugs in the Papyrus EAST-ADL, this property can not be set. Hence in the model transformation plugin, when all the allocation links are extracted from the model, it is impossible to distinguish which link belongs to which scenario. If multiple allocation objects are created, an in each object a diagram is added, the problem still won't be solved. Because there is no property in profile *Allocation* to identify each allocation object. If multiple allocation diagrams are needed to represent different states of a system, such as figure 5.6 and 5.7 illustrates the network before and after dynamic configuration, a solution is adding only allocation 5.6 in the model, after the transformation, change the allocation to 5.7 and run the transformation again, in this way the two scenarios are distinguished by the transformation plugin.
After creating hardware and function elements, the timing events and constraints can be related to HDA and FDA. In section 4.2 it is already discussed that how to set up such relations depends on modeling tools, and even with the same tool, there are various methods to realize it. One implementable solution in Papyrus EAST-ADL is illustrated.

In Papyrus EAST-ADL a flow port can only be added in a function type, not in a function prototype. All the function prototypes typed by the same function type share the same flow port. So if the dependency between the flow port and the event is set up, all the function prototypes have the same timing elements. But in FDA each prototype should have its own timing events and constraints. Any object implemented with profile *EventFunctionFlowPort* can be considered as an event of the flow port. *EventFunctionFlowPort* has a property *port*, which is used to set up the dependency between the function prototype and the event of its flow port. However because of design bug of Papyrus EAST-ADL the property *port* can not be set to its corresponding function prototype. If it can be set, even the flow port are created in the function type, each function prototype still can have its own flow port event.
One solution for set up dependency between event and function prototype is to draw an object in FDA diagram, implement the *EventFunctionFlowPort* to it, then add a dependency link between the event and the function prototype. The added dependency link will be automatically added in packages *FunctionalElements*. In this way the dependency is set up. However, although this link does exist in the model, it can not be found by the model-to-model transformation program. Another solution is to add the event directly in the function flow port. In this way the dependency between the event and the flow port is set up, but as the event is added in the function type, not the function prototype, all the function prototypes of the same type have the same flow port event. The disadvantage of this solution is each function type can only one instance if the function prototypes in FDA are designed to be with different behavior from each other. Each function type must be typed by a unique function type. Figure 5.8 illustrates the flow port event added to function type *A*.

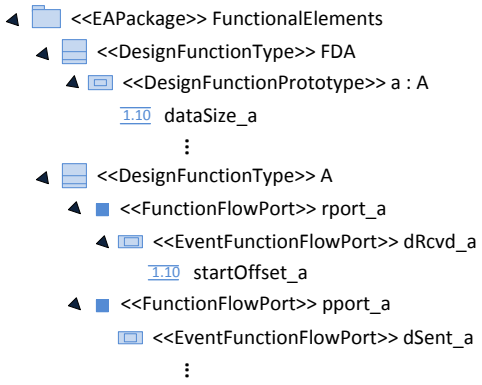

Figure 5.8.: Timing Events in Package FunctionalElements

The offset from the cycle start to the point-in-time when the function is triggered is an essential timing constraint. It indicates when a function is triggered. The data-receiving event on the rport is the beginning of function reaction. So the start offset should be related to the flow port event on the rport. The event on the *rport_a* has a

property $Defaultvalue$. This $Defaultvalue$ is set as the start offset $startOffset\_a$. Since data-receiving period of the function is decided by the the data-sending function, this period is omitted in the use-case. All the data-sending periods of the function prototypes are identical. The reason is that if a function flow port is connected to multiple functions and and each function sends out data with different periods, the corresponding data-receiving period on each function connector will be different. So on the flow port each function connector needs its own data-received event and periodic constraint. Then there will be multiple events added in one flow port. In order to set up the dependency between the periodic constraint and the function connector, there must be a method to distinguish every event and its corresponding function connector. However in profile *FunctionConnector* there is no such a property to set up relation with a flow port event. And in profile *Event-FuncitonFlowPort* there is no such a property to set up relation with a function connector. So if there are multiple events added in one flow port, it is impossible for the model-to-model transformation to identify which event is for which function connector.

Take the use-case for example. Function prototype h receives data from prototype f and g. If f and g send out data with different periods, the flow port of h must receive the data with different periods too, the flow port $rport\_h$ of function type H should contain two flow port events, one is for receiving data from function connector $f2h$, the other for $g2h$. But when the events are extracted from the EAST-ADL model, there is no way to distinguish which is for $f2h$ and which is for $g2h$. Hence in the use-case, all the function prototypes receive and send data with the same period. Figure 5.9 illustrates the timing elements of function prototype a and $application1$ in package $TimingElements$.

Package $timing\_App1$ is the reaction constraint of $application1$ which contains a, b, c and d. In figure 5.6 it is illustrated that data comes from a and finally reaches c and d respectively. So there are two reaction constraints for $application1$. $rDelay\_a2c$ is the reaction delay for data from a to c and $rDelay\_a2d$ is the reaction delay for data from a to d. $trigger_App1$ is the trigger of $timing\_App1$. Profile *FunctionTrigger* is implemented. The system is time-triggered and data from a is generated periodically, so the property *triggerPolicy* is set to *TIME*. $trigger_App1$ triggers the reactions from a to c and from a to d. One way to relate the reaction constraints to the trigger is adding $rDelay\_a2c$ and $rDelay\_a2d$ in $trigger\_App1$ and implement profile *ReactionConstraint*. It will be more convenient to identify them during the model-to-model transformation. $timing\_App2$ is the reaction constraint of $application2$ which contains e, f, g and h.

In $TimingElements$ the object $Time$ indicates all the data in this model are of type time value. The profile *Quantity* is implemented. $micro\_sec$ defines the unit of the time values is micro second. The profile *Unit* is implemented. The size of the data is decided by the function prototype which generates the data. Hence the data size can be added to function prototype in FDA. In figure 5.8 function prototype a has a property $Defaultvalue$. It is set as the data size $dataSize\_a$.

Some timing elements are created in package $HardwareElements$, as shown in figure

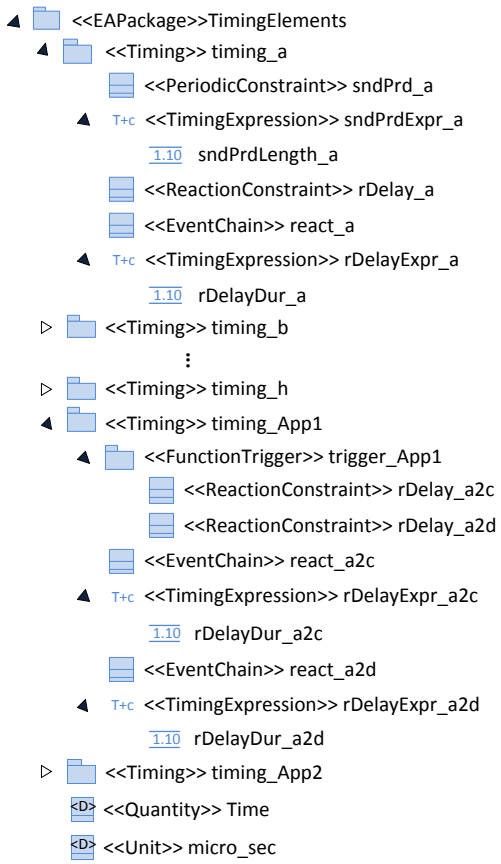

Figure 5.9.: Timing Events and Constraints of Use-Case

5.10. The reaction delay from the point-in-time when the data is sent out from the pport to the point-in-time when the data frame is waiting in the communication driver is dependent on the ECU property, it can be defined in ECU prototype. Take ECU prototype $ecu1$ as an example. It has a property $Defaultvalue$. It is set as the ECU reaction delay $delay_ecu1$. The bus speed is defined in the porter connector $TTEthernet$. The property $busSpeed$ is used to set the network speed.

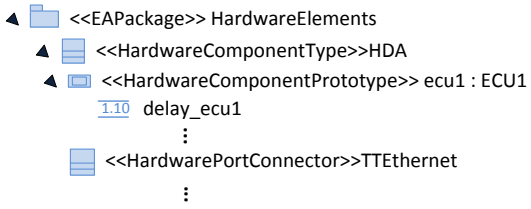

Figure 5.10.: Timing Elements in Package HardwareElements

5.2. Time Values of Use-Case Model

The *TimingExpressions* added in the model provides time values for events. In this section the values are set for simulation. Figure 5.11 illustrates the reaction delays of *application*1. The start offset and the reaction delay of the function prototypes are also set. The cycle start is considered as the original point. It is set to $0us$, all the values are set based on this original point. The values of *application*2 is shown in figure 5.12.

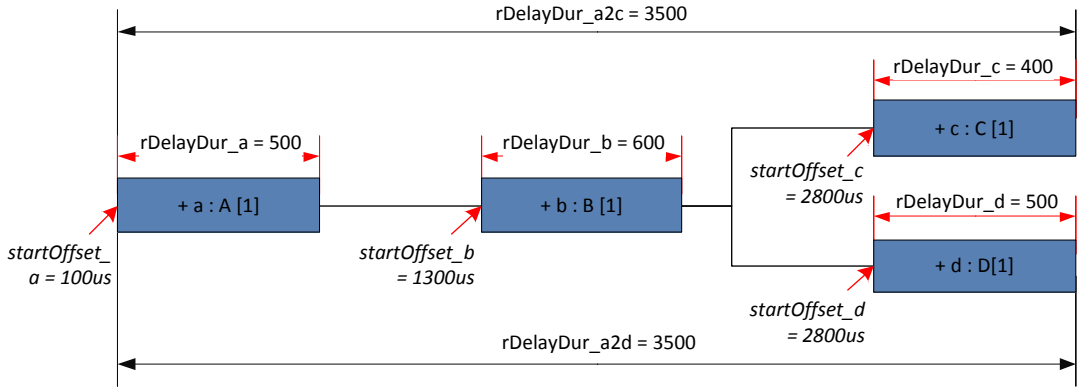

Figure 5.11.: Time Values Of Application1

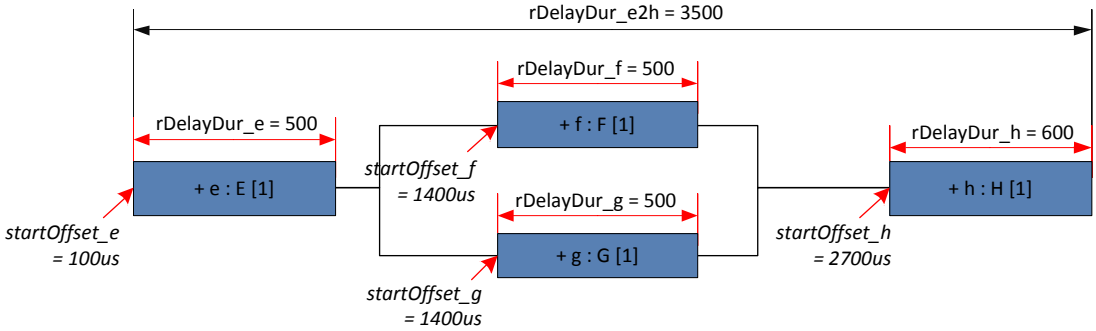

Figure 5.12.: Time Values Of Application2

The data-receiving period is omitted in the model. All the function prototypes have identical data-sending period, the value is set to $4000us$. The send delay and the receive delay between the flow port and the communication driver are the delay of ECU. It is assumed all the ECUs have identical delay, which is set to $50us$. The payload of the data frame is set to $1500Byte$, which is the maximum payload of an Ethernet frame. It is assumed that each function prototype generates the data which size is the maximum of the Ethernet frame size. Hence each data frame carries the data belonging to only one function prototype. The speed of the network bus is set to $100Mbps$.

Based on the function-to-hardware allocation, there are two simulation scenarios. Figure 5.6 illustrates $Senario_1$ and figure 5.7 illustrates $Senario_2$. The position of each $transWindow$ on the time axis in $Senario_1$ is shown in figure 5.13.

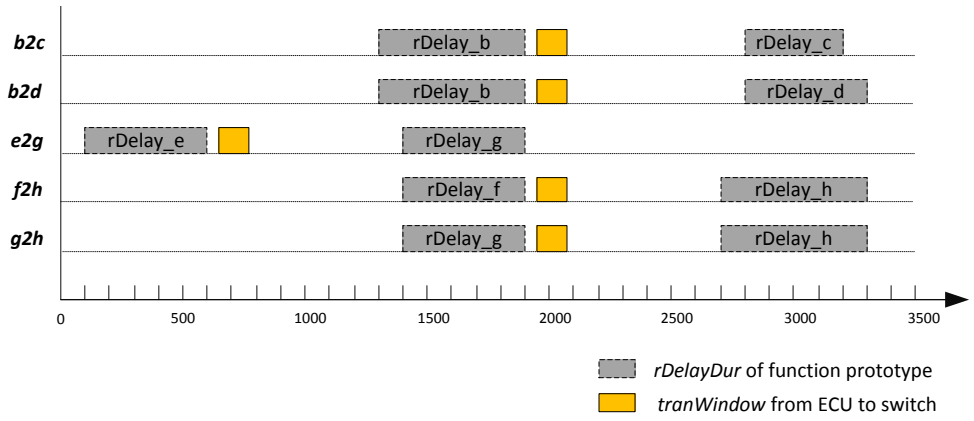

Figure 5.13.: Transmitting Windows on Time Axis with Unset Transmitting Offsets

The $transWindow$ on each connection is the transmission delay of the data frame from the ECU prototype to the switch. The $rDelayDur\_b$ ends at $1300 + 600 = 1900us$, $rDelayDur\_f$ and $rDelayDur\_g$ end at $1400 + 500 = 1900us$. The ECU delay is $50us$ on all the ECUs. Hence the data frames on communication $b2c$, $b2d$, $f2h$ and $g2h$ are ready to be sent at the same time, which is $1900 + 50 = 1950us$. The $transOffset$s of the communications still keep the initial value $0us$. So their $transWindow$s are overlapping. In order to avoid transmission collisions, three of the four communications must reset their $transOffset$s.
Equation (3.7) and (3.17) are used to calculate the $transWindow$ and the $transOffset$. In table 5.1 list the calculated values in $Senario_1$. Column $Comm$ lists the *inter-ECU communication*s, $frameReady$ lists the point-in-time when the data frame is ready for transmission, $transOffset$ lists the transmitting offsets of each transmission, $transStart$ lists the point-in-time when the frame transmission starts, $fromSw$ lists the point-in-time when the data is transmitted from the switch to the destination and $frameRcvd$ lists the point-in-time when the data frame reaches the destination, the time unit is us:

Comm	frameReady	transOffset	transStart	fromSw	frameRcvd
e2g	650	0	650	800	920
f2h	1950	0	1950	2100	2220
g2h	1950	132	2082	2232	2352
b2c	1950	264	2214	2364	2484
b2d	1950	396	2346	2496	2616

Table 5.1.: Time Values of the First Scenario

Because $b2d$ and $b2c$ have larger data receiving deadlines than $f2h$ and $g2h$, so their data-receiving deadlines can bear larger $transOffsets$. Between $f2h$ and $g2h$, the $transOffset\_g2h$ is chosen to be reset. According to equation (3.7), the data size is $1500Byte$, the bus speed is $100Mbps$, so the length of the $transWindow$ is $(1500 * 8)/(100 * 10^6) = 0.00012s$, i.e. $120us$. According to equation (3.17), the $transOffset\_g2h$ is $120 * 1.1 = 132us$. By $b2c$ it is $132 * 2 = 264us$, by $b2d$ it is $132 * 3 = 396us$. The $transOffset\_e2g$ keeps the initial value, i.e. $0us$. Hence the start of the $transWindow\_e2g$ is $650 + 0 = 650us$, by $f2h$ it is $1950 + 0 = 1950$, by $g2h$ it is $1950 + 132 = 2082$, by $b2c$ it is $1950 + 264 = 2214$ and by $b2d$ it is $1950 + 396 = 2346$.
When the data frame has reached the switch, it is transmitted further to the destination. In the transformation plugin, it is set the transmitting delay in the switch is 25% of the $transWindow$, i.e. after the data has reached the switch, it waits for $120*0.25 = 30us$. So the expected transmitting time of $e2g$ is $650+120+30 = 800us$, by $f2h$ it is $1950 + 120 + 30 = 2100us$, by $g2h$ it is $2082 + 120 + 30 = 2232us$, by $b2c$ it is $2214 + 120 + 30 = 2364us$ and by $b2d$ it is $2346 + 120 + 30 = 2496us$.
From the switch to the destination, the transmission will cost at least another $120us$. The expected arrival time of $e2g$ at the destination will be $800 + 120 = 920us$, by $f2h$ it will be $2100 + 120 = 2220us$, by $g2h$ it will be $2232 + 120 = 2352us$, by $b2c$ it will be $2364 + 120 = 2484us$ and by $b2d$ it will be $2496 + 120 = 2616us$.
Figure 5.14 shows the position of $transWindows$ in $senario1$ after the $transOffsets$ are reset.

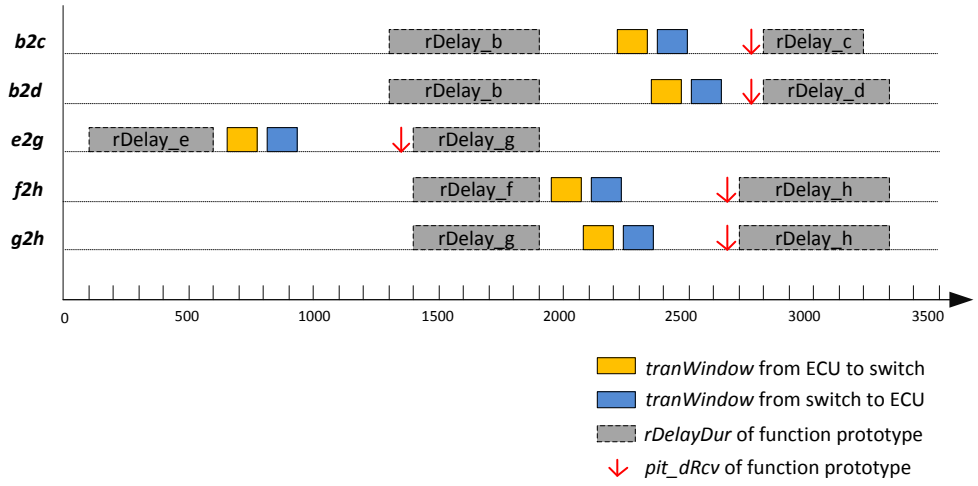

Figure 5.14.: Transmitting Windows of the First Scenario

The red arrows in the figure are the data-received deadlines on the destination function prototypes, before which data frame must arrive. The yellow $transWindows$ are the transmitting windows for the data frames transmitted from the ECU prototypes to the switch. The blue ones are the transmitting windows for the frames from the switch further transmitted to the destination ECU prototypes.

One thing to notice is the yellow $transWindow$s on each communication does not overlap with other yellow $transWindow$s, and the blue $transWindow$ on each communication does not overlap with other blue $transWindow$s, but there are overlapped $transWindow$s of different colors. In section 5.3 the simulation will illustrate if transmission collisions occur due to this overlapping.
In $Senario_2$, function prototype e is moved to $ecu1$ and f is moved to $ecu3$. $ecu2$ is empty and won't transmit any data frames. The new emerged *inter-ECU communication* is $e2f$. The $frameReady\_e2f$ is $100+500+50 = 650us$, which is the same with $frameReady\_e2g$. The $transOffset\_e2g$ is chosen to be reset. The calculated values in $Senario_2$ are listed in table 5.2, the time unit is us:

Comm	frameReady	transOffset	transStart	fromSw	frameRcvd
e2f	650	0	650	800	920
e2g	650	132	782	932	1052
f2h	1950	0	1950	2100	2220
g2h	1950	132	2082	2232	2352
b2c	1950	264	2214	2364	2484
b2d	1950	396	2346	2496	2616

Table 5.2.: Time Values of the Second Scenario

The length of the $transWindow$ is $120us$. According to equation (3.17), the value of $transOffset\_e2g$ is $120 * 1.1 = 132us$. The $transOffset\_e2f$ keeps the initial value, i.e. $0us$. The start of the $transWindow\_e2f$ is $650 + 0 = 650us$, by $e2g$ it is $650 + 132 = 782$. The delay in the switch is $42us$, the expected further transmitting time of $e2f$ is $650 + 120 + 30 = 800us$, by $e2g$ it is $782 + 120 + 30 = 932us$. The $transWindow$ from the switch to the destination ECU is $120us$. The expected arrival time of $e2f$ will be $800 + 120 = 920us$, by $e2g$ it will be $932 + 120 = 1052us$. The position of other $transWindow$s keep unchanged on the time axis. Figure 5.15 shows the position of $transWindow$s in $Senario_2$ after the $transOffset\_e2f$ and $transOffset\_e2f$ are reset.

5.3. Analysis and Evaluation of Simulation Result

After all the values of the model are set, the ned and ini files can be created by the transformation plugin. The generated network is based on the network structure ned file, as shown in figure 5.16. There are four ECUs connected by a TTEthernet switch. The parameters in the ini files are set automatically by the plugin based on the values in the model.

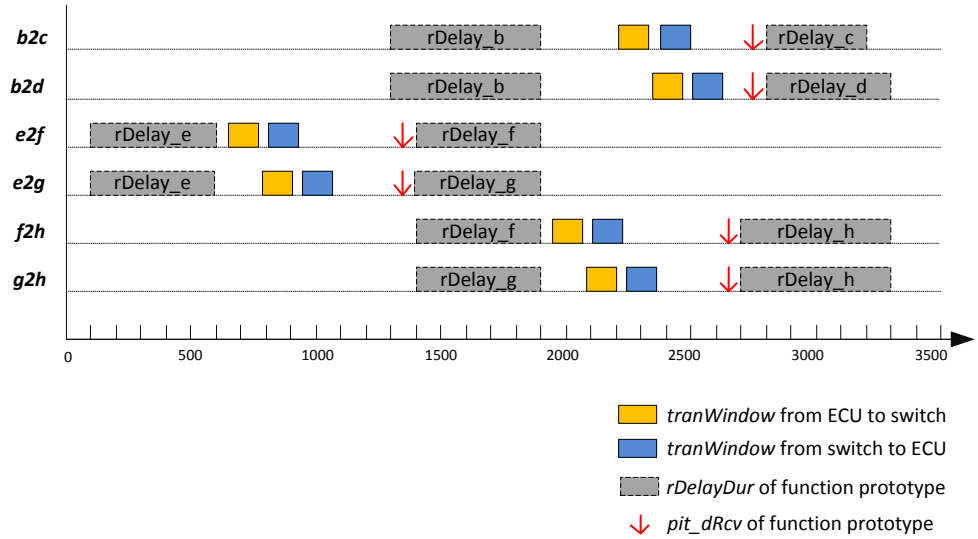

Figure 5.15.: Transmitting Windows of the Second Scenario

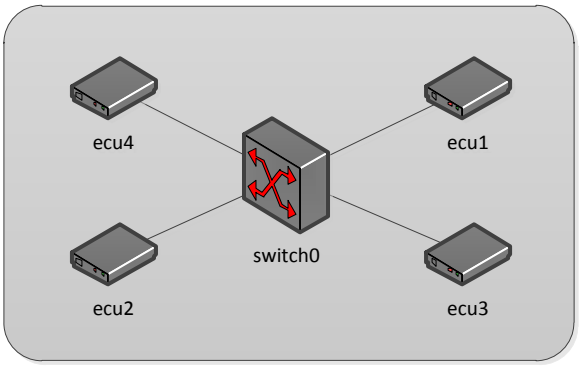

Figure 5.16.: Generated Network in OMNeT++

In order to examine the accuracy of the generated parameters, traffic *b2c* is checked. It starts from *ecu*1 and ends at *ecu*4. So *ecu*1.*ini*, *ecu*4.*ini* and *TTSwitch.ini* are checked. The part of the Xtend template which generates the data-sending parameters of ECU is:

```
FOR  buffer : ecu.buffers
       # Defines TTDoubleBuffer buffer.name
       **. ecu . name . buffer . name .destination_gates = "FOR g : buffer. destination_gatesg ,
             ENDFOR  "
       **. ecu . name . buffer . name .ct_id = buffer. ct_id
       **. ecu . name . buffer . name .sendWindowStart = sec_to_tick(buffer.sendWindowStartus)
       **. ecu . name . buffer . name .period = "period[buffer. ct_id]"
 ENDFOR
```

The corresponding part in *ecu*1.*ini* illustrates that the data is sent from physical port phy[0] at 2214*us*, as calculated in table 5.1. The data-sending period is *period*[3]. Although all the data-sending period of function prototypes are identical, the plugin

still identifies the period values:

```
# Defines TTDoubleBuffer b2c_buffer
**.ecu1.b2c_buffer.destination_gates = "phy[0].TTin, "
**.ecu1.b2c_buffer.ct_id = 3
**.ecu1.b2c_buffer.sendWindowStart = sec_to_tick(2214.0us)
**.ecu1.b2c_buffer.period = "period[3]"
```

The part of the Xtend template which generates the data-exchanging parameters in the switch is:

```
FOR  incoming : ttswitch.incomings
      # Defines TTIncoming incoming.name
      **. ttswitchindex . incoming . name .receive_window_start = sec_to_tick(incoming.
          receive_window_startus)
      **. ttswitchindex . incoming . name .receive_window_end = sec_to_tick(incoming.
          receive_window_endus)
      **. ttswitchindex . incoming . name .permanence_pit = sec_to_tick(incoming.permanence_pitus)
      **. ttswitchindex . incoming . name .period = "period[incoming.ct_id]"
 ENDFOR

FOR  buffer : ttswitch.buffers
      # Defines TTDoubleBuffer buffer.name
      **. ttswitchindex . buffer . name .destination_gates = "FOR g : buffer. destination_gatesg ,
           ENDFOR "
      **. ttswitchindex . buffer . name .ct_id = buffer. ct_id
      **. ttswitchindex . buffer . name .sendWindowStart = sec_to_tick(buffer.sendWindowStartus)
      **. ttswitchindex . buffer . name .period = "period[buffer.ct_id]"
 ENDFOR
```

In the corresponding part of *TTSwitch.ini* it is illustrated that the data is received by physical port phy[0], which connects *ecu*1 and sent from phy[3], which connects *ecu*4. The data is sent out from *ecu*1 at 2214*us*, the transmission duration is 120*us*, the expected point-in-time when the data arrives at the switch is $2214 + 120 = 2334us$. According to the algorithm in the plugin, the width of the window is 15% of the transmission duration, i.e. 18*us*, so the receive window should start at $2334 - 18 = 2316$us and ends at $2334 + 18 = 2352us$. In *TTSwitch.ini* the receive window starts at 2316*us* and ends at 2352*us*, as expected. The *permanence_pit* is the point-in-time when the data is 'permanenced'. It is a TTEthernet-defined concept, which means to synchronize arrived data from multiple connections. In this case *permanence_pit* is equal to *receive_window_end*, it is assumed the data is synchronized as soon as the transmitting window ends. The data-receiving period is *period*[3], which is identical to the data-sending period in *ecu*1. The data is sent out at 2364*us* from the switch as calculated in table 5.1:

```
# Defines TTIncoming b2c_incoming
**.ttswitch0.b2c_incoming.receive_window_start = sec_to_tick(2316.0us)
**.ttswitch0.b2c_incoming.receive_window_end = sec_to_tick(2352.0us)
**.ttswitch0.b2c_incoming.permanence_pit = sec_to_tick(2352.0us)
**.ttswitch0.b2c_incoming.period = "period[3]"
...
# Defines TTDoubleBuffer b2c_buffer
**.ttswitch0.b2c_buffer.destination_gates = "phy[3].TTin, "
**.ttswitch0.b2c_buffer.ct_id = 3
**.ttswitch0.b2c_buffer.sendWindowStart = sec_to_tick(2364.0us)
**.ttswitch0.b2c_buffer.period = "period[3]"
```

The part of the Xtend template which generates the data-receiving parameters in ECU is:

```
FOR  incoming : ecu.incomings
      # Defines TTIncoming incoming.name
      **. ecu . name . incoming . name .receive_window_start = sec_to_tick(incoming.
          receive_window_startus)
      **. ecu . name . incoming . name .receive_window_end = sec_to_tick(incoming.receive_window_endus
          )
      **. ecu . name . incoming . name .permanence_pit = sec_to_tick(incoming.permanence_pitus)
      **. ecu . name . incoming . name .period = "period[incoming.ct_id]"
 ENDFOR
```

In the corresponding part of *ecu*4.*ini*, the data is received by physical port phy[0]. According to the algorithm in the plugin, the total time delay from *ecu*1 to *ecu*4 through the switch is 25% of the transmission duration, i.e. $30us$. So the expected point-in-time when the data arrives *ecu*4 is $2484us$. The width of the receive window is set the same as the ECU delay, i.e. $50us$. So the window should starts at $2434us$ and ends at $3524us$. The values of *receive_window_start* and *receive_window_end* in *ecu*4.*ini* are as expected. The data-receiving period is *period*[3], identical to the data-sending period in *ecu*1:

```
# Defines TTIncoming b2c_incoming
**.ecu4.b2c_incoming.receive_window_start = sec_to_tick(2434.0us)
**.ecu4.b2c_incoming.receive_window_end = sec_to_tick(2534.0us)
**.ecu4.b2c_incoming.permanence_pit = sec_to_tick(2534.0us)
**.ecu4.b2c_incoming.period = "period[3]"
```

The simulation log of $Scenario_1$ is shown in table A.1 in appendix. The events of the first communication cycle is listed and analyzed. The complete simulation log of the first communication cycle records events from #1 to #177, in which contains much information which is not related to the network data transmission, such as network nodes internal events or synchronization messages. The events which concern the network traffic is selected and illustrated in the table. Column ID lists the event ID, $Time$ lists the point-in-time when the event occurs, the time unit is *second*, *Event* lists the content of the event.
The procedure of transmitting data from the source to the destination on a communication link can be separated into six phase:

(1) The data is generated by the traffic source application in the source ECU.
(2) The data is transmitted from the ECU's physical port on the network media.
(3) The data is received by the switch.
(4) The data is further transmitted form the switch's physical port on the network.
(5) The data is received by the physical port of the destination ECU.
(6) The data is consumed by the traffic sink application in the ECU.

In table A.1, the events of each traffic interweave with each other, which makes the data evaluation difficult. In table 5.3, the events are sorted based on the six phases. The table lists the event IDs. The traffic of communication $f2h$ is analyzed.

	e2g	f2h	g2h	b2c	b2d
Source App	16	60	61	49	50
ECU → Switch	24	73	82	98	125
Switch	25	74	90	117	146
Switch → ECU	36	89	113	140	163
ECU	37	102	129	152	167
Sink App	47	116	143	166	177

Table 5.3.: Sorted Events of the First Scenario

In event #60 the data $f2h\_buffer$ is generated by the traffic source application $app[1]$ in $ecu2$ at $1399.92us$. It is for the function f to process at $startOffset\_f$, which is set to $1400us$ in figure 5.12.
In event #73 $f2h\_buffer$ is transmitted on the network at $1950us$. According to figure 5.12 and table 5.1, $f2h\_buffer$ is processed by f at $1400us$, the reaction delay is 500, then the data is sent from the rport of f to the $ecu2$'s physical port, it costs another $50us$. The expected $twOffset\_e2g$ is $1950us$. The simulation value is as expected.
In event #74 $f2h\_buffer$ is received by $switch0$'s physical port $phy[1]$ at $2072.08us$. $twOffset\_e2g$ is $1950us$, the $transWindow$ is $120us$, so the expected arrival time at $switch0$ is $2070us$. The simulation value is $2.08us$ later. The cause of the deviation is the $transWindow$ is calculated based on the data frame load. The load is the maximum Ethernet frame load $1500Byte$. There are also the head and foot of the frame, together the size of the frame is $1528Byte$, i.e. $12208bit$. The bus speed is $100Mbps$, so the real value of $transWindow$ in the simulation is $122.08us$.
In event #89 $f2h\_buffer$ is sent out from $switch0$ to $ecu3$ by the physical port $phy[2]$ at $2100us$, as the $fromSw\_f2h$ calculated in table 5.1. After the switch has received the data, it needs some time to relay it to the destination. In the transformation plugin, the length of this waiting time is set to 25% of a $transWindow$. The expected $transWindow$ is $120us$, so switch waiting time is $30us$. In the simulation, the time delay between #74 and #89 is $2100 - 2072.08 = 27.92us$, which is $2.08us$ shorter than the expected waiting time.
In event #102 $f2h\_buffer$ is received data by $ecu4$'s physical port at $2222.08us$. The $fromSw\_f2h$ is $2100us$, the $transWindow$ is $120us$, so the expected data-received time is $2220us$, the simulation value is $2.08us$ later.
In event #116 $f2h\_buffer$ is sent from the physical port to the traffic sink application $app[0]$ at $2270us$. By event #111 at $2232us$ (which is not listed in table A.1), $app[0]$ creates the empty buffer for the coming data. The time delay between #102 and #116 is $2270 - 2222.08 = 47.92$. This is the ECU delay, which is $2.08us$ shorter than the set valued $50us$. At $startOffset\_h$, $f2h\_buffer$ will be processed by h and transmitted further.
One thing to notice is there are transmitting windows overlapping on the time axis. For example, the $transWindow$ of $g2h$ from $ecu3$ to $switch0$ starts at $2082us$ and

ends at $2202us$, the $transWindow$ of $f2h$ from $switch0$ to $ecu4$ starts at $2100us$ and ends at $2200us$. But there is no collision on the network. The reason is the topology of the network. $ecu3$ and $ecu4$ connects to the switch with their own links, as long as there is no simultaneous transmissions on the same link, there won't be any collision.
The scheduling algorithm in Chapter 3 considers all the network nodes share the same communication link, it will schedule any $transWindow$s which overlap on the time axis. Hence in figure 5.14, even if $f2h$ and $g2h$ start from different ECUs, the $transWindow$ of $f2h$ is still moved. The switch topology actually adds more shared communication links in the network, that allows more network nodes to transmit data at the same time, the switch is in charge of media access control. Comparing the bus topology, the switch topology is more flexible, and it can provide higher speed, but it also becomes more complex. The scheduling algorithm is designed to fit any TDMA network regardless the topology. Hence it has to omit the advantage of switch-based network in order to be compatible with bus-based network. The traffic order of $Scenario_1$ is shown in table 5.4.

ID	Time	Traffic	Data
#24	0.00065	ecu2 → ttswitch0	e2g_buffer
#36	0.0008	ttswitch0 → ecu3	e2g_buffer
#73	0.00195	ecu2 → ttswitch0	f2h_buffer
#82	0.002082	ecu3 → ttswitch0	g2h_buffer
#89	0.0021	ttswitch0 → ecu4	f2h_buffer
#98	0.002214	ecu1 → ttswitch0	b2c_buffer
#113	0.002232	ttswitch0 → ecu4	g2h_buffer
#125	0.002346	ecu1 → ttswitch0	b2d_buffer
#140	0.002364	ttswitch0 → ecu4	b2c_buffer
#163	0.002496	ttswitch0 → ecu3	b2d_buffer

Table 5.4.: Traffic Order of the First Scenario

In $Scenario_2$, function prototype e is move to $ecu1$ and f is moved to $ecu3$. $ecu2.ini$ becomes empty. The new *inter-ECU communication*is $e2f$, which is overlapping with $e2g$. The data transmitting and receiving time of $e2g$ are reset. The simulation log of $Scenario_2$ is shown in table A.2 in appendix. The events from #1 to #207 belong to the first communication cycle, the ones which concern the network traffic is listed and analyzed. The events are sorted in table 5.5.

	e2f	e2g	f2h	g2h	b2c	b2d
Source App	17	18	91	92	81	82
ECU → Switch	29	38	103	112	128	155
Switch	30	46	104	120	147	174
Switch → ECU	45	65	119	143	170	193
ECU	54	69	132	159	182	197
Sink App	68	79	147	173	196	207

Table 5.5.: Sorted Events of the Second Scenario

The traffic of communication $e2f$ change the timing of $e2g$. Further more, because f and g are both in $ecu3$, it is necessary to move the $transWindow$ of $g2h$, otherwise there will be transmission collision from $ecu3$ to $switch0$. The traffic order of $Scenario_2$ is shown in table 5.6.

ID	Time	Traffic	Data
#29	0.00065	ecu1 → ttswitch0	e2f_buffer
#38	0.000782	ecu1 → ttswitch0	e2g_buffer
#45	0.0008	ttswitch0 → ecu3	e2f_buffer
#65	0.000932	ttswitch0 → ecu3	e2g_buffer
#103	0.00195	ecu3 → ttswitch0	f2h_buffer
#112	0.002082	ecu3 → ttswitch0	g2h_buffer
#119	0.0021	ttswitch0 → ecu4	f2h_buffer
#128	0.002214	ecu1 → ttswitch0	b2c_buffer
#143	0.002232	ttswitch0 → ecu4	g2h_buffer
#155	0.002346	ecu1 → ttswitch0	b2d_buffer
#170	0.002364	ttswitch0 → ecu4	b2c_buffer
#193	0.002496	ttswitch0 → ecu3	b2d_buffer

Table 5.6.: Traffic Order of the Second Scenario

6. Conclusion and Outlook

6.1. Conclusion of the Work

This thesis provides a solution of real-time network reconfiguration for the self-adaptive automotive system. The self-adaptive functionality increases the flexibility and reliability of the automotive system. It can be realized based on the current industry specification and system architecture, hence it is cost and time efficient to extend a non-self-adaptive system with the such a functionality, meanwhile keep it compatible with other systems which do not support self-adaptive functionality.
In this thesis the function communication is modeled based on the AUTOSAR timing extension. Depending on the function-to-hardware allocation the concepts of *intra-ECU communication* and *inter-ECU communication* are developed. The behavior of the communication can be expressed by the events and their timing constraints. The timing events on the VFB level and the system level are defined, which are used to configure the network access. Based on the communication model a scheduling algorithm for network media access is developed. The algorithm is based on EDF. The concept of the *scheduling axis* is defined, all the transmitting windows are scheduled on the this axis. The scheduling algorithm contains two parts, the *local scheduling* and the *global scheduling*. The *local scheduling* is the basic scheduling algorithm, it moves the new emerged transmitting window on the *scheduling axis* to find idle time slot, other transmitting windows keep unchanged. The *global scheduling* is an iteration of *local scheduling*. Not only the new transmitting windows, but also the scheduled windows can be moved in order to make full use of the idle time slots. The scheduling algorithm developed in the thesis is a simple algorithm. The advantage is that it is easy to implement. A mature algorithm implemented in the industry is relatively complex, and it is time-cost to implement it in the transformation plugin. And it is impossible to avoid bugs in the implementation. When the simulation result is not as expected, it might be caused either by the communication model or by the implementation of the scheduling algorithm. The latter one will disturb the improvement of the communication model. Hence a simple algorithm is helpful to evaluate the function communication model.
The function communication modeling concept is implemented in EAST-ADL modeling, and OMNeT++ simulation library is integrated to evaluation the performance of the network. A model transformation plugin is created, which transform the EAST-ADL model to ned and ini files, and the scheduling algorithm is implemented to set the network parameters of the simulation scenarios. The plugin provides a solution for an early phase validation during designing a system. A use-case is created

to evaluate the communication model and the model transformation, the simulation log illustrates the result is as expected.

6.2. Outlook of the Future Work

The thesis provides a solution to model automotive network communication on the design level of EAST-ADL, based on this model, based on which the network access scheduling can be analyzed and improved. In the future more timing elements can be added in the model to increase the accuracy of the model. The EAST-ADL model contains timing elements on the VFB level, i.e. the events and constraints of the function flow ports. The function is treated as a black box, its reaction delay is considered as a given condition, which is estimated by the function designer. This estimated value is a worst case reaction delay, in reality, the reaction delay of a function is usually smaller than the worst case delay. In AUTOSAR timing extension the timing elements on the SW-C level is defined, which deals with the internal behavior of atomic software components. On the SW-C level the function can be treated as a white box, the execution time and interruptions can be foreseen hence the reaction delay will be more accurately estimated. The problem is, the function on the design level is not equal to software component. Depending on the functionality, a function can be mapped either to an atomic software component or to a composition. If the function is mapped to an atomic software component, the SW-C level events can be added to it. If the function is mapped to a composition, the communication inside the function is component-to-component, which is still the VFB level events. In such a situation, the function-to-software allocation must be defined and the SW-C level events must be added to each software component, then the reaction delay of a function can be estimated.
Because timing events on the system level are not available in EAST-ADL, in order to define the point-in-time when an event occurs in ECU's communication driver, the ECU delay is defined. The data is transmitted from the pport, goes through the BSW layer and reaches the communication driver. This process is controlled by the ECU operating system, transmitting data is one of the tasks in the ECU's task list. The value of ECU delay is hard to estimate. The current value is a worst case delay and it is probably much higher than the actual delay. In AUTOSAR timing extension there are timing elements defined on the BSW level and the ECU level. The BSW level timing treats the internal behavior of a BSW module, the ECU level timing treats the software deployment, the basic software and ECU interaction. Such events can be added to the ECU prototypes, hence to make the estimation more accurate. This solution can only improve the estimation of the delay, it is still difficult to pin-point when the data frame is ready or is received at the communication driver. A better solution is to model the communication on the implementation level of EAST-ADL, i.e. in AUTOSAR.
As shown in Chapter 5 there is problem to set up dependency between the flow port and the event. Current solution relates the event to the flow port in the function

type. Hence all the function prototypes which are typed by the same function type have the same timing event and constraint. So if each function prototype has different timing elements from others, it must be typed by a unique function type.
All the function prototypes have the same data transmitting period. If each function prototype has its own data-sending period, then the data-receiving periods will be different, so each function connector needs its own data-received event and periodic constraint. In EAST-ADL profile *FunctionConnector* there is no such a property to set an event, and in profile *EventFuncitonFlowPort* there is also no such a property to set a function connector. It is impossible to distinguish the dependency between the event and the function connector. If there are multiple events added in the flow port, it is impossible to identify which event is for which function connector in the model-to-model transform. Until now there is no better solution for these two problems.
In the transformation plugin, because all the function prototypes have the same data-sending period in the EAST-ADL model, hence the equation (3.1) for calculating the communication length is omitted. In the scheduling function of the plugin, when the data-received deadline or the application deadline is missed, the program throws an exception and the transformation stops. In the next step, a better warning method for the user should be created.
One missing reaction delay in the model is the transformation delay. It definitely costs some time to move a software component and reconfigure the system. On the run-time, if this delay is too long, the application reaction deadline may be missed. The current plugin creates the two simulation scenarios independently. First allocation 5.6 is used for creating the scenario before the software component is moved, after the transformation, the allocation is changed to 5.7, the transformation is executed once more to create the scenario after the moving. A better solution is set a transformation delay in the model, which is an estimated value, and create two allocation diagrams. The plugin transforms the model according to the first allocation, wait for the transformation delay, then transforms the model according to the second allocation, the transformation delay is used in the second transformation to validate if the application deadline is missed.
Another work in the future can be mapping the EAST-ADL model to AUTOSAR model. One advantage to model a system in on the EAST-ADL design level is the system is designed in the function view, not the software view. The designer can on the one hand focus on the functionality of the system, and on the other hand decide the function-to-hardware allocation, which makes it possible to evaluate the system performance in the early phase. However, the disadvantage is the designer may need some more concrete information which can not be provided by EAST-ADL. The timing elements in EAST-ADL are only available on the VFB level. Although EAST-ADL also provides profile *AUTOSAREvent* for events on the implementation level, it is still inconvenient to implement such profile, because there are different AUTOSAR events, it impossible to distinguish them.
The advantage to model the system on the implementation level is that AUTOSAR does not only provide events on the VFB level, but also provides the events on the

system level. To the communication model in Chapter 3, the events which receive and send data frame on the network are available. The disadvantage is the system is designed in the software view, the designer must be clear with the function-to-software allocation. Once the functionality is changed, one or more software design is affected. In Appendix B the function communication model is created by Papyrus EAST-ADL and Artop respectively, the mapping relations between the two models are illustrated.

A. Simulation Log of Two Scenarios

ID	Time	Event
#16	0.00009992	hda.ecu2.app[0] on 'API Scheduler Task Event'
#24	0.00065	hda.ecu2.phy[0].mac on 'e2g_buffer'
#25	0.00077208	hda.ttswitch0.phy[1].mac on 'e2g_buffer'
#36	0.0008	hda.ttswitch0.phy[2].mac on 'e2g_buffer'
#39	0.00092208	hda.ecu3.phy[0].inControl on 'e2g_buffer'
#47	0.00097	hda.ecu3.e2g_buffer on 'e2g_buffer'
#49	0.00129992	hda.ecu1.app[0] on 'API Scheduler Task Event'
#50	0.00129992	hda.ecu1.app[1] on 'API Scheduler Task Event'
#60	0.00139992	hda.ecu2.app[1] on 'API Scheduler Task Event'
#61	0.00139992	hda.ecu3.app[1] on 'API Scheduler Task Event'
#73	0.00195	hda.ecu2.phy[0].mac on 'f2h_buffer'
#74	0.00207208	hda.ttswitch0.phy[1].mac on 'f2h_buffer'
#82	0.002082	hda.ecu3.phy[0].mac on 'g2h_buffer'
#89	0.0021	hda.ttswitch0.phy[3].mac on 'f2h_buffer'
#90	0.00220408	hda.ttswitch0.phy[2].mac on 'g2h_buffer'
#98	0.002214	hda.ecu1.phy[0].mac on 'b2c_buffer'
#102	0.00222208	hda.ecu4.phy[0].mac on 'f2h_buffer'
#113	0.002232	hda.ttswitch0.phy[3].mac on 'g2h_buffer'
#116	0.00227	hda.ecu4.f2h$_b uffer on `f2h_buffer'
#117	0.00233608	hda.ttswitch0.phy[0].mac on 'b2c_buffer'
#125	0.002346	hda.ecu1.phy[0].mac on 'b2d_buffer'
#129	0.00235408	hda.ecu4.phy[0].mac on 'g2h_buffer'
#140	0.002364	hda.ttswitch0.phy[3].mac on 'b2c_buffer'
#143	0.002402	hda.ecu4.g2h_buffer on 'g2h_buffer'
#146	0.00246808	hda.ttswitch0.phy[0].inControl on 'b2d_buffer'
#152	0.00248608	hda.ecu4.phy[0].mac on 'b2c_buffer'
#163	0.002496	hda.ttswitch0.phy[2].mac on 'b2d_buffer'
#166	0.002534	hda.ecu4.b2c_buffer on 'b2c_buffer'
#167	0.00261808	hda.ecu3.phy[0].mac on 'b2d_buffer'
#177	0.002666	hda.ecu3.b2d_buffer on 'b2d_buffer'

Table A.1.: Simulation Log of the First Scenario

ID	Time	Event
#17	0.00009992	hda.ecu1.app[0] on 'API Scheduler Task Event'
#18	0.00009992	hda.ecu1.app[1] on 'API Scheduler Task Event'
#29	0.00065	hda.ecu1.phy[0].mac on 'e2f_buffer'
#30	0.00077208	hda.ttswitch0.phy[0].mac on 'e2f_buffer'
#38	0.000782	hda.ecu1.phy[0].mac on 'e2g_buffer'
#45	0.0008	hda.ttswitch0.phy[2].mac on 'e2f_buffer'
#46	0.00090408	hda.ttswitch0.phy[0].mac on 'e2g_buffer'
#54	0.00092208	hda.ecu3.phy[0].mac on 'e2f_buffer'
#65	0.000932	hda.ttswitch0.phy[2].mac on 'e2g_buffer'
#68	0.00097	hda.ecu3.e2f_buffer on 'e2f_buffer'
#69	0.00105408	hda.ecu3.phy[0].mac on 'e2g_buffer'
#79	0.001102	hda.ecu3.e2g_buffer on 'e2g_buffer'
#81	0.00129992	hda.ecu1.app[2] on 'API Scheduler Task Event'
#82	0.00129992	hda.ecu1.app[3] on 'API Scheduler Task Event'
#91	0.00139992	hda.ecu3.app[2] on 'API Scheduler Task Event'
#92	0.00139992	hda.ecu3.app[3] on 'API Scheduler Task Event'
#103	0.00195	hda.ecu3.phy[0].mac on 'f2h_buffer'
#104	0.00207208	hda.ttswitch0.phy[2].mac on 'f2h_buffer'
#112	0.002082	hda.ecu3.phy[0].mac on 'g2h_buffer'
#119	0.0021	hda.ttswitch0.phy[3].mac on 'f2h_buffer'
#120	0.00220408	hda.ttswitch0.phy[2].mac on 'g2h_buffer'
#128	0.002214	hda.ecu1.phy[0].mac on 'b2c_buffer'
#132	0.00222208	hda.ecu4.phy[0].mac on 'f2h_buffer'
#143	0.002232	hda.ttswitch0.phy[3].mac on 'g2h_buffer'
#147	0.00233608	hda.ttswitch0.phy[0].mac on 'b2c_buffer'
#155	0.002346	hda.ecu1.phy[0].mac on 'b2d_buffer'
#159	0.00235408	hda.ecu4.phy[0].mac on 'g2h_buffer'
#170	0.002364	hda.ttswitch0.phy[3].mac on 'b2c_buffer'
#174	0.00246808	hda.ttswitch0.phy[0].mac on 'b2d_buffer'
#182	0.00248608	hda.ecu4.phy[0].mac on 'b2c_buffer'
#196	0.002534	hda.ecu4.b2c_buffer on 'b2c_buffer'
#197	0.00261808	hda.ecu3.phy[0].mac on 'b2d_buffer'
#193	0.002496	hda.ttswitch0.phy[2].mac on 'b2d_buffer'
#207	0.002666	hda.ecu3.b2d_buffer on 'b2d_buffer'

Table A.2.: Simulation Log of the Second Scenario

B. EAST-ADL Model to Artop Model Mapping

The relations between EAST-ADL and AUTOSAR is discussed in Chapter 4. This part of appendix illustrates how to map an EAST-ADL model to an Artop model. Artop (AUTOSAR Tool Platform) [43] is an ideal AUTOSAR modeling tool. It is based on the Eclipse platform, including EMF and Sphinx. It provides functionality for tools that are used to design and configure AUTOSAR systems and ECUs. The common Functionality of modeling is provided by the Eclipse Sphinx project. The architecture of Artop is shown in figure B.1. The Artop applied in the thesis is Version 4.2.1.

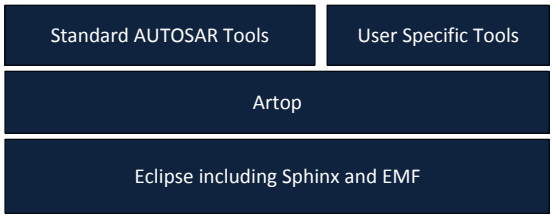

Figure B.1.: Architecture of Artop

The disadvantage of EAST-ADL modeling is that its timing elements are only available on the VFB level. It is impossible to describe the behavior on the system level of AUTOSAR in EAST-ADL. AUTOSAR is on the implementation level of EAST-ADL, which provides timing elements on five different views. To map the function communication model in chapter 3, the designer must be clear with the function-to-software allocation. Once the functionality is changed, one or more software design is affected. The timing events and constraints of the communication model can be mapped to Artop model as shown in figure B.2.
VFBTiming and SystemTiming components are added in ARPackage *Timing*. Every timing element in EAST-ADL can find its corresponding element in VFBTiming. Because the EAST-ADL Timing conforms to AUTOSAT timing extensions. In Artop an object is created with the profile, in EAST-ADL the profile must be implemented manually after an object is created. Because EAST-ADL modeling is on the more abstract level, which also provides flexibility, the user can define his or her own profile if needed. Besides the VFB level, Artop also provides elements on the SW-C, BSW and system level. The events which can not be defined in the EAST-ADL can be defined in Artop. Such as the data frame receiving and transmitting events.

Figure B.2.: Mapping Relations between ESAT-ADL Model and Artop Model

Bibliography

[1] AUTOSAR. Technical overview, 2011.

[2] AUTOSAR. Layered software architecture, 2014.

[3] AUTOSAR. Virtual functional bus, 2014.

[4] AUTOSAR. Specification of timing extensions, 2014.

[5] Matthias Werner. 03 scheduling, echtzeitsystem, 2014.

[6] Matthias Werner. 07 kommunikation, echtzeitsysteme, 2014.

[7] EAST-ADL Association. EAST-ADL domain model specification, 2013.

[8] ATESST. ATESST, 2010.

[9] AUTOSAR. AUTOSAR home, 2014.

[10] SafeAdapt. Definition of use cases and scenarios for safe adaptation, March 2014.

[11] Richard Anthony, Achim Rettberg, Dejiu Chen, Isabell Jahnich, Gerrit de Boer, and Cecilia Ekelin. Towards a dynamically reconfigurable automotive control system architecture. In *Embedded System Design: Topics, Techniques and Trends*, pages 71–84. Springer, 2007.

[12] D. Kohler. A practical implementation of an IEEE1588 supporting ethernet switch. In *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication, 2007. ISPCS 2007*, pages 134–137, October 2007.

[13] Hung-Manh Pham, Sebastien Pillement, and D. Demigny. Reconfigurable ECU communications in autosar environment. In *2009 9th International Conference on Intelligent Transport Systems Telecommunications,(ITST)*, pages 581–585, October 2009.

[14] Jos Lus Nunes, Joo Carlos Cunha, Raul Barbosa, and Mrio Zenha-Rela. Using partial dynamic FPGA reconfiguration to support real-time dependability. In *Proceedings of the 13th European Workshop on Dependable Computing*, EWDC '11, pages 107–108, New York, NY, USA, 2011. ACM.

[15] Dong Yin, Deepak Unnikrishnan, Yong Liao, Lixin Gao, and Russell Tessier. Customizing virtual networks with partial FPGA reconfiguration. *SIGCOMM Comput. Commun. Rev.*, 41(1):125–132, January 2011.

[16] Sbastien Saudrais and Khaled Chaaban. Automatic relocation of AUTOSAR components among several ECUs. In *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, CBSE '11, pages 199–204, New York, NY, USA, 2011. ACM.

[17] Purnendu Sinha. Dynamic task-level reconfiguration in automotive software architectures. In *Proceedings of the 6th India Software Engineering Conference*, ISEC '13, pages 35–44, New York, NY, USA, 2013. ACM.

[18] R. Casado, A. Bermudez, J. Duato, F.J. Quiles, and J.L. Sanchez. A protocol for deadlock-free dynamic reconfiguration in high-speed local area networks. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):115–132, February 2001.

[19] Daniel Ldtke and Dietmar Tutsch. Lossless static vs. dynamic reconfiguration of interconnection networks in parallel and distributed computer systems. In *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC '07, pages 717–724, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[20] Frank Olaf Sem-Jacobsen and Olav Lysne. Topology agnostic dynamic quick reconfiguration for large-scale interconnection networks. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 228–235, Washington, DC, USA, 2012. IEEE Computer Society.

[21] P. Heinrich, D. Eilers, R. Knorr, M. Koniger, and B. Niehoff. Autonomous parameter and schedule configuration for TDMA-based communication protocols such as FlexRay. In *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1241–1246, November 2011.

[22] Robert Brendle, Thilo Streichert, Dirk Koch, Christian Haubelt, and Jrgen Teich. Dynamic reconfiguration of FlexRay schedules for response time reduction in asynchronous fault-tolerant networks. In Uwe Brinkschulte, Theo Ungerer, Christian Hochberger, and Rainer G. Spallek, editors, *Architecture of Computing Systems ARCS 2008*, number 4934 in Lecture Notes in Computer Science, pages 117–129. Springer Berlin Heidelberg, January 2008.

[23] year = 2007 pages = 79–87 Reichelt, {Stephan and Schmidt, Karsten and Gesele, Frank and Seidler, Nils and Hardt, Wolfram. Nutzung von FlexRay als zeitgesteuertes automobiles bussystem im AUTOSAR-umfeld. In *Mobilitt und Echtzeit*. GI Gesellschaft fr Informatik e.V.

[24] Jakob Axelsson and Avenir Kobetski. On the conceptual design of a dynamic component model for reconfigurable AUTOSAR systems. *SIGBED Rev.*, 10(4):45–48, December 2013.

[25] Basil Becker, Holger Giese, Stefan Neumann, Martin Schenck, and Arian Treffer. Model-based extension of AUTOSAR for architectural online reconfiguration. In *Proceedings of the 2009 International Conference on Models in Software Engineering*, MODELS'09, pages 83–97, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] Jens Halbig, Andr Windisch, Patrick Kingsbury, Norbert Oswald, and Wolfram Hardt. Integration of realtime decision-making in time-triggered software architectures for certifiable autonomous unmanned systems. In *Proceedings of 2011 2nd International Conference on Instrumentation Control and Automation (ICA 2011)*, pages 219–224. IEEE Computer Society, November 2011.

[27] FlexRay. FlexRay communications system protocol specification version 2.1, 2005.

[28] CiA. CAN in automation (CiA): Time-triggered CAN, 2014.
[29] Juergen Schwager. Real-time-ethernet in der industrieautomation, 2013.
[30] Max Felser. Real-time ethernet industry prospective. PROCEEDINGS OF THE IEEE, VOL. 93, NO.6, June 2005.
[31] IEEE 802.1 Task Group. IEEE 802.1 AV bridging task group, 2013.
[32] IEEE 802.1 Task Group. IEEE 802.1 time-sensitive networking task group, 2012.
[33] TTTech. TTEthernet deterministic ethernet network - TTTech, 2014.
[34] AUTOSAR. Concept global time synchronization, 2014.
[35] NIST US Department of Commerce. IEEE 1588, 2014. IEEE 1588.
[36] EAST-ADL Association. EAST-ADL association, 2013.
[37] OMNet++. OMNeT++ network simulation framework, 2014.
[38] Sirius. Sirius, 2014.
[39] TTTech. TTEthernet specification, 2008.
[40] Papyrus. Papyrus, 2014.
[41] INET. INET framework, 2014.
[42] CoRE. CoRE simulation models for real-time networks, 2014.
[43] Artop. Artop - AUTOSAR tool platform user group, 2014.

Nomenclature

ACC	Adaptive Cruise Control
Artop	AUTOSAR Tool Platform
AW	Acception Window
AUTOSAR	AUTomotive Open System ARchitecture
BE	best-effort
BSW	Basic SoftWare
DNR	Dynamic Network Reconfiguration
DSR	Dynamic Software Reconfiguration
EAST-ADL	EAST Architecture Description Language
ECU	Electronic Control Unit
EDF	Earliest Deadline First
E/E	Electric/Electronic
EMT	Eclipse Modeling Tools
FAA	Function Analysis Architecture
FIFO	first-in-first-out
FDA	Function Design Architecture
HDA	Hardware Design Architecture
LCM	Least Common Multiple
MAC	Media Access Control
NIT	network idle time
RC	rate-constrained
SW-C	SoftWare Component
TDMA	Time Division Multiple Access
TT	time-triggered
VFB	Virtual Functional Bus